UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA EN INFORMÁTICA

Implementación de una aplicación para almacenamiento y visualización de imágenes utilizando tecnologías *cloud computing*

Autor: Gonzalo Suárez Llorente
Tutor: Pablo Basanta Val

Implementación de una aplicación para almacenamiento y visualización de imágenes utilizando tecnologías *cloud computing*

Gonzalo Suárez Llorente

20 de marzo de 2013

Resumen

Los sistemas de información son elementos básicos para la gestión de empresas y organizaciones. Cuando el número de estos sistemas y su complejidad son elevados los costes pueden incrementarse de forma drástica. En los últimos años, algunas organizaciones han optado por soluciones basadas en las tecnologías *cloud computing* con las que han mejorado la calidad del servicio y han ahorrado costes.

El objetivo de este proyecto es diseñar e implementar una aplicación web sencilla para gestionar fotografías similar a *Flickr* empleado dichas tecnologías. Para ello se utilizarán diversos servicios en la nube como la plataforma *Heroku* donde se ejecutará la aplicación y los servicios auxiliares *Amazon S3*, *MongoHQ* y *Google Image Charts* que facilitan alojamiento masivo de archivos, almacenamiento de datos estructurados y generación de gráficos respectivamente.

Finalmente, se realizan una serie de pruebas bajo diferentes condiciones para evaluar su rendimiento utilizando las herramientas *Siege* y *New Relic*.

Abstract

Information systems are basic elements for business and corporation management. As the number and complexity of these systems increase, the cost of managing them might get too high to be affordable. During the last years, some corporations have adopted solutions based on *cloud computing* technologies which provide better quality service and cost savings.

The main purpose of this project is to design and implement a simple web application similar to *Flickr* that allows a user to manage and view photographies taking advantage of such technologies. In order to do it, several *cloud* services are used such as *Heroku* platform on which the application will be deployed and executed and complementary services such as *Amazon S3*, *MongoHQ* and *Google Image Charts* which provide file storage, structured data storage and graphics generation respectively.

Finally, application will be benchmarked under different scenarios to check its performance by means of the tools *Siege* and *New Relic*.

Agradecimientos

A mis padres Inma y Andrés, que se han preocupado siempre por ofrecerme lo mejor y que me han dado todo su amor, comprensión, ánimo y cariño, especialmente valiosos en los momentos duros, cuando las cosas no salen como uno espera.

A mis abuelos, que siempre se interesan por la marcha de su nieto en los estudios.

A Mónica por su inmensa paciencia durante estos últimos meses.

A los autores de las referencias, cuyo trabajo me ha inspirado y servido de gran ayuda.

Y a mi tutor Pablo Basanta, por su entusiasmo, interés y ánimos mostrados durante el proyecto.

A todos, muchas gracias.

Índice general

Índice de figuras

Índice de tablas

Parte I

Introducción

Capítulo 1

Introducción al proyecto

1.1. Introducción

Hoy en día, la mayoría de medianas y grandes organizaciones dependen de sistemas de información para poder ser gestionadas de forma eficiente. Tradicionalmente, muchos de estos sistemas requieren de una gran cantidad y variedad de recursos *hardware* y *software* para ser implementados. Su diseño, implementación, instalación, configuración y mantenimiento a menudo implican la coordinación de un gran número de expertos. Cuando en una organización el número de estos sistemas crece hasta varias decenas, su coste puede verse incrementado de forma muy notable.

Es por esto que, durante la última década, un conjunto de tecnologías englobadas bajo el término *cloud computing* han despertado el interés de muchas empresas que buscan, por un lado, modernizar y mejorar sus sistemas de información y por otro reducir los costes que estas suponen.

Si tomamos, por ejemplo, el omnipresente sistema de correo electrónico y analizamos los elementos básicos que son necesarios para poder ofrecer este servicio nos daremos cuenta que pueden ser muy numerosos y heterogéneos.

- *Hardware:* sistemas de alimentación ininterrumpida y de refrigeración, equipos de red y servidores entre otros.
- *Software:* sistemas operativos, bases de datos y servidores de nombres de dominio (DNS) y de correo (SMTP e IMAP/POP3).

Es un sistema complejo por el gran número de elementos distintos que aglutina y su mantenimiento a menudo resulta muy costoso. Por ello, algunas empresas como Zinkia [1], BBVA [2] o FON [3], ya no mantienen servidores de correo en sus instalaciones y han comenzado a utilizar soluciones basadas en *cloud computing* como *Google Apps for Business*[1].

[1] http://www.google.com/enterprise/apps/business/

El propósito del presente trabajo es diseñar una aplicación web (figura 1.1), similar a *Flickr*[71], que permita al usuario almacenar, buscar, clasificar y visualizar sus fotografías e implementarla mediante una serie de tecnologías *cloud computing* o servicios en la nube. Para llevar a cabo esta tarea, se han marcado una serie de objetivos más específicos que se enumerarán en el siguiente apartado.

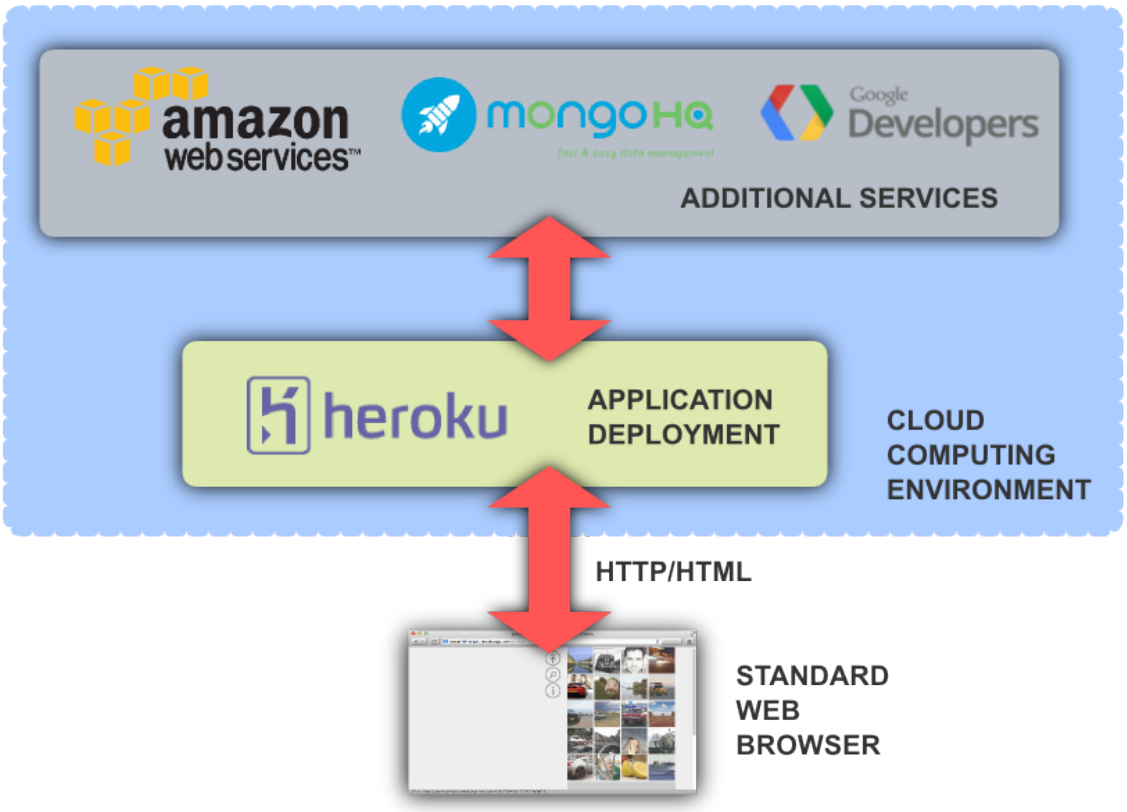

Figura 1.1: Arquitectura de la aplicación desarrollada

1.2. Objetivos

- Exponer una definición de *cloud computing* y explicar cuáles son sus características esenciales así como sus modelos de servicio y despliegue más habituales.
- Presentar las herramientas, plataformas y servicios que serán necesarios para la implementación de la aplicación.
- Diseño, implementación y puesta en producción de la aplicación.
- Evaluación del funcionamiento y rendimiento de la aplicación.

- Análisis de los resultados de la evaluación anterior.

1.3. Estructura de la memoria

A lo largo de esta memoria se documenta estructuralmente el trabajo realizado durante el proyecto. Se ha dividido en nueve capítulos y dos apéndices.

1.3.1. Capítulos

- **Capítulo 1: Introducción al proyecto.** La memoria comienza con una introducción al trabajo realizado abordando la problemática a la que muchas organizaciones se enfrentan cuando tienen que desarrollar y mantener un gran número de sistemas de información. Posteriormente se enuncian los objetivos del presente proyecto y finalmente se detalla, en el último apartado, la estructura de la memoria.

- **Capítulo 2: *Cloud Computing.*** En este capítulo se define qué es *cloud computing* y se enumeran y explican las características esenciales que deben ofrecer, así como sus modelos de servicio y de despliegue más comunes.

- **Capítulo 3: *Platform as a Service.*** En este capítulo se realiza una breve introducción a las plataformas como servicios en la nube (*PaaS*). Posteriormente, se detallan los aspectos tecnológicos más importantes de la plataforma *Heroku* sobre la que se realizará el despliegue de la aplicación objeto del presente proyecto.

- **Capítulo 4: *Software as a Service.*** En este capítulo se presentan los servicios en la nube (*SaaS*) *Amazon S3*, *MongoHQ* y *Google Image Charts* sobre los que depende la aplicación desarrollada.

- **Capítulo 5: Diseño.** Este capítulo presenta los aspectos más relevantes del diseño de la aplicación. En él se definen los requisitos funcionales, y se especifica la interacción entre los diversos elementos de la aplicación.

- **Capítulo 6: Implementación.** A lo largo de las secciones de este capítulo se abordan los detalles técnicos de la implementación más relevantes como el lenguaje y *framework* utilizados, bibliotecas e interacción entre los servicios en la nube entre otros.

- **Capítulo 7: Despliegue y control.** En este capítulo se aborda cómo preparar y configurar la máquina de los desarrolladores para poder implementar, desplegar y controlar aplicaciones en la plataforma *Heroku*.

- **Capítulo 8: Evaluación.** En este capítulo se describen las pruebas realizadas sobre la aplicación y se analizan los resultados.

- **Capítulo 9: Conclusiones y futuras líneas de trabajo**. El último capítulo recoge las conclusiones que se han obtenido tras concluir el presente trabajo y cuáles son las áreas más interesantes sobre las que cabría profundizar.

1.3.2. Apéndices

- **Apéndice A: Presupuesto** Este apéndice contiene en detalle los cálculos del esfuerzo, tiempo y coste tanto en recursos humanos como materiales que ha supuesto el desarrollo del proyecto. Incluye igualmente la definición de las tareas y su situación y duración en el tiempo, dando lugar al calendario del proyecto.
- **Apéndice B: Metodología *twelve-factor app*** Este apéndice recoge las doce reglas descritas por la metodología *twelve-factor app* que facilitan el desarrollo de aplicaciones web modernas, escalables y fácilmente mantenibles.

1.3.3. Glosario

En el glosario se incluyen definiciones de acrónimos y términos poco comunes utilizados a lo largo del documento y cuyo conocimiento es recomendable para una buena compresión de éste.

1.3.4. Bibliografía

Por último, la bibliografía recoge todas aquellas referencias mencionadas a lo largo de la memoria y que pueden servir al lector tanto para comprobar las fuentes de información como para ampliar detalles sobre algún aspecto en particular. Adicionalmente, también se incluye otra documentación consultada y relacionada con el proyecto.

Parte II

Situación actual

Capítulo 2

Cloud computing

2.1. Introducción

Las tecnologías en la nube o *cloud computing* se refieren fundamentalmente a productos y servicios ofrecidos a través de una red de comunicaciones, habitualmente Internet. Independientemente de su propósito, todos ellos comparten una serie de características como pueden ser que son accesibles desde cualquier punto de una red de comunicaciones, que el servicio ofrecido puede ser medido o que la capacidad del servicio se ajusta a las necesidades del consumidor entre otros. Por otro lado, los servicios en la nube se pueden ofrecer en base a tres niveles de servicio (infraestructura, plataforma y aplicación) y de despliegue (nube pública, nube privada, nube comunitaria o una mezcla de todas ellas).

2.2. Definición

El Instituto Nacional de Normas y Tecnología (NIST, por sus siglas en inglés) define *cloud computing* [4] como un modelo mediante el cual es posible acceder a través de red (de forma sencilla, desde cualquier lugar y según a una serie de necesidades) a un conjunto compartido de recursos informáticos (como redes, servidores, almacenamiento, aplicaciones y servicios) que pueden estar disponibles para su uso rápidamente con un mínimo esfuerzo de gestión o interacción por parte del proveedor de estos servicios.

Cloud computing requiere para su implementación de una infraestructura específica denominada *cloud infrastructure* y está formada por dos capas. La primera, *physical layer* o capa física agrupa todos los elementos *hardware* que son necesarios para soportar los servicios como servidores, almacenamiento, alimentación ininterrumpida, refrigeración y equipamientos de red. La segunda, *abstraction layer* o capa de abstracción consiste en el *software* desplegado sobre la primera y proporciona las cinco características esenciales que todos los servicios *cloud computing* comparten y que son descritos en la siguiente sección. Conceptualmente, como se muestra en la figura 2.1, la capa de abstracción se asienta sobre la capa física.

Figura 2.1: Capas en *cloud infrastructure*

2.3. Características esenciales

On-demand self-service

El consumidor puede poner a su disposición una serie de servicios o recursos informáticos (como tiempo de cómputo en un servidor o almacenamiento masivo en red) de forma automática según los necesite, sin requerir intervención humana previa con cada uno de los proveedores.

Broad network access

Los servicios o recursos están disponibles a través de la red (típicamente Internet) y son accesibles a través de mecanismos estándar que favorecen su uso desde plataformas muy diversas, como teléfonos móviles, *tablets*, portátiles o estaciones de trabajo y servidores.

Resource pooling

Los recursos informáticos del proveedor están agrupados para atender a varios consumidores a la vez (según un modelo *multi-tenant*) donde varios recursos físicos y virtuales son dinámicamente asignados de acuerdo a las necesidades de cada consumidor. El cliente generalmente no tiene control o conocimiento sobre la localización exacta de los recursos que usa pero puede, en ocasiones, especificar una ubicación de forma abstracta como un país, una región o un determinado centro de datos. Ejemplos de recursos son almacenamiento, tiempo de cómputo, memoria o ancho de banda.

Rapid elasticity

Los recursos requeridos por un servicio pueden ser asignados y liberados rápidamente, en muchos casos de forma automática, para adaptarse a la demanda. Para el consumidor

del servicio, los recursos disponibles se muestran, a menudo, ilimitados y se pueden adecuar a las necesidades en cualquier momento y cantidad.

Measured service

Los sistemas basados en *cloud computing* automáticamente controlan y optimizan el uso de recursos utilizando algún mecanismo de medida apropiado al tipo de servicio ofrecido como tiempo de cómputo, ancho de banda, almacenamiento, número de transacciones o cuentas de usuario activas entre otros. Así, es posible visualizar, controlar y confeccionar informes sobre los recursos consumidos proporcionando transparencia tanto para el proveedor como para el consumidor del servicio.

2.4. Modelos de servicio

Software as a Service (SaaS)

Bajo este modelo, el consumidor utiliza las aplicaciones que un proveedor pone a su disposición y que se ejecutan sobre una *cloud infraestructure* o infraestructura en la nube. Estas aplicaciones son accesibles desde diferentes tipos de dispositivos (ordenadores, teléfonos móviles y *tablets*) bien mediante el uso de clientes ligeros como un navegador web, bien mediante clientes pesados o programas específicos. El consumidor no controla o administra la infraestructura subyacente que incluye equipos de red, servidores, sistemas operativos, almacenamiento o incluso las propias aplicaciones que utiliza, a excepción de algunos aspectos relacionados con la configuración de usuarios sobre estas últimas.

Platform as a Service (PaaS)

En esta ocasión, el servicio ofrecido al consumidor es la capacidad para desplegar sobre la infraestructura del proveedor aplicaciones realizadas por él mismo o compradas a terceros. El consumidor no controla o administra la infraestructura subyacente que incluye equipos de red, servidores, sistemas operativos o almacenamiento pero tiene control sobre las aplicaciones desplegadas y los parámetros que controlan el entorno para su ejecución. Generalmente, las aplicaciones deben ser desarrolladas utilizando lenguajes, bibliotecas, servicios y herramientas soportados por la plataforma.

Infrastructure as a Service (IaaS)

El último modelo proporciona al consumidor la capacidad para proveerse de recursos de cómputo, almacenamiento, redes, y otros elementos fundamentales de computación sobre los que puede desplegar y ejecutar cualquier *software* incluyendo sistemas operativos y aplicaciones. El consumidor no controla o administra la infraestructura subyacente, pero sí tiene control sobre sistemas operativos, almacenamiento, aplicaciones desplegadas, y

posiblemente control limitado sobre algunos aspectos de la red de comunicaciones como *firewalls* o cortafuegos.

2.5. Modelos de despliegue

Private cloud

La infraestructura es proporcionada exclusivamente para uso de una sola organización que comprende varios consumidores (por ejemplo, los diferentes departamentos de una empresa). Puede ser propiedad de la organización y estar dirigida y operada por ésta, por un tercero, o por alguna combinación de ambas y puede residir físicamente fuera o dentro de las instalaciones de la organización.

Community cloud

La infraestructura es proporcionada para uso exclusivo de un grupo específico de consumidores cuyas organizaciones comparten preocupaciones similares (por ejemplo, misión, requisitos de seguridad, políticas varias). Puede ser propiedad de una o varias de las organizaciones y estar dirigida y operada por éstas, por un tercero, o por alguna combinación de ambas y puede residir físicamente fuera o dentro de las instalaciones de la organización.

Public cloud

La infraestructura es proporcionada para uso del público en general. Puede ser propiedad de una compañía o institución educativa o gubernamental y estar dirigida y operada por éstas, por un tercero, o por alguna combinación de ambas y reside físicamente en las instalaciones del proveedor.

Hybrid cloud

La infraestructura es una composición de dos o más infraestructuras (privada, comunitaria o pública) que están separadas pero que se combinan juntas mediante tecnología estándar o propietaria permitiendo potabilidad de datos y aplicaciones. Esto hace posible, por ejemplo, *cloud bursting* o balanceo la carga entre infraestructuras.

2.6. Conclusiones

Las tecnologías *cloud computing* presentan una interesante alternativa a la hora de componer la arquitectura de las aplicaciones. Algunas de sus principales ventajas son su capacidad casi infinita y costes reducidos. Por otro lado, los diferentes modelos de servicio permiten determinar el nivel de abstracción y de responsabilidad adecuados al proyecto.

Por último, los modelos de despliegue ofrecen la posibilidad de determinar el tipo de nube sobre la que se desea instalar las aplicaciones.

Capítulo 3

Platform as a Service

3.1. Introducción

En el primer capítulo presentamos algunos casos prácticos de empresas que ya reducen notablemente sus costes en sistemas de información utilizando *Google Apps for Business*, una solución *software as a service (SaaS)* para la gestión de correo electrónico, calendarios y almacenamiento y compartición de documentos. Hoy en día, existe una gran oferta de aplicaciones *SaaS* que tratan de cubrir todas las necesidades posibles que empresas y organizaciones necesitan. A continuación se listan algunas de las más relevantes:

- **Kyriba[5]:** Tesorería y análisis de riesgos.
- **Sales Cloud[6]:** Ventas y *CRM*.
- **Service Cloud[7]:** Atención al cliente mediante redes sociales y canales tradicionales.
- **Yammer[8]:** Redes sociales privadas para empresas y organizaciones.
- **FreshBooks[9]:** Facturación y contabilidad.

En ocasiones, puede ocurrir que ninguna de las opciones *SaaS* disponibles en el mercado cumpla con los requisitos que demanda una organización, bien sea por precio elevado, bien sea por funcionalidad limitada. En estos casos, será necesario producir una aplicación a medida, algo que tradicionalmente tiene un coste elevado ya que todas las fases del desarrollo deben ser asumidas por la organización. La inversión necesaria en recursos humanos y en equipos podría hacer inviable el proyecto.

Las soluciones basadas en *platform as a service (PaaS)*, ofrecen simplificar el proceso de desarrollo de *software* eliminando así algunos de los costes más importantes. Bajo este modelo de servicio, las aplicaciones se construyen utilizando las herramientas, lenguajes y bibliotecas (habitualmente *open source*) compatibles con el proveedor y finalmente se despliegan, ejecutan y controlan sobre la infraestructura de éste. Los gastos de adquisición y

mantenimiento de *hardware* (equipos de red, servidores, y dispositivos de almacenamiento masivo) se reducen a cero.

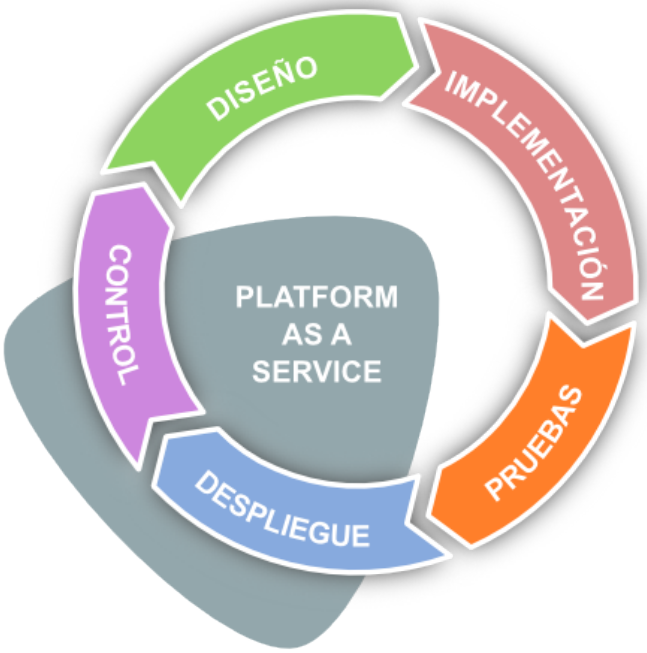

Figura 3.1: Ciclo de desarrollo y *PaaS*

3.2. Características

La decisión sobre qué solución *PaaS* utilizar es crucial y debe tomarse atendiendo a los factores que se exponen a continuación.

3.2.1. Lenguajes de programación

Algunos proveedores *PaaS* ofrecen soluciones que permiten la ejecución de aplicaciones realizadas con lenguajes o entornos de programación específicos. Este es el caso, por ejemplo, de *Engine Yard*[10], que a día de hoy solo soporta:

- Ruby[11]
- PHP[12]
- Node.js[13] (Soporte anunciado el 21 de Agosto de 2012)[14]

En cambio, otros como *Heroku*[15] diseñan su plataforma para poder ejecutar aplicaciones implementadas en cualquier lenguaje o entorno de programación. Esto es posible gracias a una capa de abstracción denominada *buildpacks*[16]. A día de hoy, *Heroku* mantiene varios *buildpacks* oficiales que soportan las siguientes tecnologías:

- Ruby[17]
- Node.js[18]
- Clojure[19]
- Python[20]
- Java[21]
- Gradle[22]
- Grails[23]
- Scala[24]
- Play[25]

La comunidad de usuarios de *Heroku* además proporciona varios *buildpacks* que permiten utilizar lenguajes y *frameworks* no soportados oficialmente por el proveedor como:

- Common Lisp[26]
- Elixir[27]
- Erlang[28]
- Go[29]
- JRuby[30]
- Lua[31]
- Perl[32][33]
- R[34]
- Silex[35]

3.2.2. Resolución de dependencias

Prácticamente cualquier aplicación tiene dependencias, bien sobre bibliotecas propias del entorno de programación, bien sobre bibliotecas realizadas por terceros. Estas bibliotecas, además, pueden ser de dos tipos: nativas o no nativas.

Algunos proveedores *PaaS* solo autorizan el uso de un conjunto y tipo específico de bibliotecas. En cambio, otros como *OpenShift*[36] o *Heroku*, permiten a los desarrolladores utilizar mecanismos propios del entorno de programación elegido para enumerar cualquier número de dependencias que serán posteriormente resueltas de forma automática por el proveedor durante la fase de despliegue de la aplicación. Por ejemplo, en *Ruby* es habitual describir las dependencias utilizando un fichero denominado `Gemfile` y gestionarlas mediante la herramienta *Bundler*.

3.2.3. Integración con servicios de terceros

Los proveedores *PaaS* proporcionan un entorno donde poder desplegar, ejecutar y controlar aplicaciones, pero estas habitualmente requieren de servicios adicionales, como persistencia de datos y otros. En entornos *cloud computing* es muy común conectar aplicaciones con servicios proporcionados por terceros (*software as a service*) para suplir estas necesidades.

El proveedor *Heroku* permite integrar de forma muy sencilla aplicaciones con algunos productos *SaaS* a través de sus *add-ons*. Por ejemplo, si se requiere que una aplicación

realice persistencia de datos sobre una base de datos relacional, *Heroku* proporciona *add-ons* para conectar con los siguientes productos *SaaS*:

- Heroku Postgres (PostgreSQL)[37]
- JustOneDB (PostgreSQL)[38]
- ClearDB MySQL Database (MySQL)[39]
- AmazonRDS (MySQL, Oracle, SQL Server)[40]
- Xeround[41]

El proveedor *PaaS* no siempre proporciona mecanismos que simplifican la conexión con el producto *SaaS* que requiere la aplicación que se está desarrollando pero ello no significa que no pueda ser utilizado.

3.2.4. Despliegue y control de las aplicaciones

Un aspecto muy importante que debe evaluarse a la hora de seleccionar una plataforma u otra son los mecanismos de despliegue y control de las aplicaciones. La fase de despliegue debe ser sencilla e intuitiva para que los desarrolladores puedan poner en los entornos de prueba y producción nuevas versiones del código fuente de forma rápida y segura. De igual forma, la plataforma debe proporcionar mecanismos para poder conocer cómo está funcionando la aplicación mediante registros claros así como un modelo que permita el escalado horizontal de la aplicación cuando la demanda de recursos aumenta.

Algunas plataformas como *Heroku* o *Engine Yard* manejan un repositorio *Git* con el código de la aplicación. Cada vez que un desarrollador incorpora código en el repositorio central se inicia un proceso de despliega la aplicación.

El proceso de control y escalado horizontal de las aplicaciones varia entre las diversas plataformas. Por ejemplo, *Heroku* dispone de un modelo de procesos (explicado con detalle en la sección 3.3.3) que permite la división de la aplicación en procesos con responsabilidades específicas. Cada uno de estos procesos es instanciado tantas veces como se necesite para adecuarse a la demanda (escalado horizontal).

3.2.5. Coste

La gran mayoría de proveedores *PaaS* ofrecen la posibilidad de utilizar su plataforma de forma gratuita (modo de evaluación) con algún tipo de limitación en tiempo o rendimiento. Esto es un aspecto muy interesante pues hace posible la puesta en marcha de prototipos sin hacer ningún desembolso.

Sin embargo, lo normal es que una aplicación no pueda ser ejecutada en producción de forma indefinida haciendo uso del modo de evaluación. Por ello, es conveniente conocer el uso de recursos que realiza la aplicación, de qué forma lo cuantifica el proveedor y que

coste genera.

Algunos aspectos sobre los que los proveedores *PaaS* se basan para configurar sus tarifas son:

- Ancho de banda (subida, bajada o ambos)
- Espacio de almacenamiento
- Tiempo de *CPU*
- Número de procesos
- Memoria

El proveedor *Heroku* calcula el coste por aplicación teniendo en cuenta los siguientes tres aspectos:

- Número y tipo de *dyno* (Obligatorio, al menos 1 *web dyno*).
- Tipo de base de datos (No obligatorio)
- Add-ons (No obligatorio)

Heroku automáticamente proporciona a cada aplicación 750 horas de *dynos* al mes. Esto significa que una aplicación que solo tiene un *web dyno* consume 720 horas cada mes, por lo que puede funcionar sin coste de forma indefinida.

3.3. La plataforma *Heroku*

3.3.1. Introducción

Heroku nació a mediados del año 2007 como una de las primeras plataformas en la nube y permitía la ejecución de aplicaciones programadas en *Ruby* y basadas en la biblioteca *Rack*[42]. Desde entonces, ha evolucionado añadiendo soporte para otros muchos lenguajes y *frameworks* (como *Java*, *Node.js*, *Scala* y *Clojure* entre otros muchos) e integración con servicios de terceros mediante *add-ons*. Su importancia dentro la industria es notable y sus principales competidores son:

- AWS Elastic Beanstalk[43]
- Engine Yard
- Google App Engine[44]
- OpenShift

- Windows Azure[45]

Desde finales de 2010 *Heroku* es propiedad de *Salesforce.com Inc*, una de las empresas pioneras en el desarrollo y comercialización de *SaaS* para empresas. En las siguientes secciones se detallan varios aspectos de su arquitectura y funcionamiento ya que esta es la plataforma sobre la cual se desplegará la aplicación objeto del presente proyecto.

3.3.2. *Stacks*

Un *stack* en la plataforma *Heroku* es un entorno completo sobre el cual se despliegan y controlan aplicaciones y está definido por un sistema operativo y un entorno de ejecución junto con sus correspondientes bibliotecas. Los *stacks* actualmente soportados se listan en la tabla 3.1.

Stack	*Operating System*	MRI[1] 1.8.6	REE[2] 1.8.6	MRI 1.9.2	Node.js	Clojure	Java	Python	Scala
Argent Aspen	Debian Etch 4.0	•							
Badious Bamboo	Debian Lenny 5.0		•	•					
Celadon Cedar	Ubuntu 10.04			•	•	•	•	•	•

Tabla 3.1: Diferentes *stacks* de la plataforma *Heroku*

Los entornos *Aspen* y *Bamboo* están diseñados solo para aplicaciones *Ruby* y *frameworks* como *Rails* o *Sinatra*. En estos momentos, el entorno por defecto y único disponible para desplegar aplicaciones nuevas es *Cedar*. Es más flexible que sus antecesores, ofrece soporte para varios lenguajes de programación, mecanismos de introspección y resistencia a la erosión. Su diseño tiene en cuenta las directrices de la metodología *Twelve-Factor App* (véase apéndice B) para el desarrollo, despliegue y control de aplicaciones ofrecidas como servicio.

Es posible conocer el *stack* sobre el que está desplegada una aplicación mediante el comando `heroku stack --app [APPNAME]`[3].

[1] Matz's Ruby Interpreter
[2] Ruby Enterprise Edition
[3] Este comando es parte del paquete *Heroku Toolbelt*. Véase la sección 7.2 para obtener más información.

```
heroku stack --app myapp
=== myapp Available Stacks
  bamboo-mri-1.9.2
  bamboo-ree-1.8.7
  cedar
* aspen-mri-1.8.6 (deprecated)
```

En el ejemplo, la aplicación `myapp` está desplegada sobre *Aspen*, el primer *stack* de la plataforma *Heroku* y ya obsoleto. En estos casos, el proveedor recomienda migrar a *Cedar*[4].

3.3.3. Modelo de procesos

El modelo de procesos de UNIX es una simple y potente abstracción para la ejecución de programas en el lado del servidor. Trasladado a las aplicaciones web, este modelo proporciona un mecanismo ideal para dividir la carga total de trabajo y escalarla horizontalmente a lo largo del tiempo. El *stack Cedar* de *Heroku* usa este modelo de procesos para procesos de tipo *web*, *worker* y otros muchos.

Dyno manifold

Es posible arrancar servicios como MySQL, Apache o Memcached directamente desde una consola de comandos. Esta práctica puede ser válida en entornos locales de desarrollo pero en un entorno de producción estos servicios deben ser ejecutados como procesos controlados (*managed processes*). Un proceso controlado debe ejecutarse de forma automática cuando el sistema operativo arranca y debe reiniciarse cuando falla o muere por alguna razón.

En los entornos tradicionales de servidores, el sistema operativo proporciona un gestor de procesos. Por ejemplo, en *OS X* se *launchd* y en *Ubuntu* es *upstart*. En la plataforma *Heroku*, el componente *dyno manifold* es el gestor de procesos distribuidos.

Aplicación del modelo a las aplicaciones web

Un demonio (*server daemon*) como *memcached* tiene un único punto de entrada lo que significa que para arrancarlo se invoca un solo comando. Las aplicaciones web sin embargo, típicamente tienen dos o más puntos de entrada. Cada uno de ellos se puede decir que es un tipo de proceso.

Una aplicación *Rails* suele tener dos tipos de procesos: un proceso web (compatible con *Rack* como *Webrick*, *Mongrel* o *Thin*) y un proceso *worker* para ejecutar trabajos en

[4]https://devcenter.heroku.com/articles/cedar-migration

segundo plano (mediante el uso de alguna biblioteca de colas como *Delayed Job* o *Resque*). Por ejemplo:

Tipo de proceso	Comando
web	`bundle exec rails server`
worker	`bundle exec rake jobs:work`

Tabla 3.2: Tipos de procesos en una aplicación tradicional *Rails*

Es necesario definir los diferentes tipos de procesos para cada aplicación. *Procfile*[46] es un formato para definirlos y *Foreman*[47] es una herramienta que facilita su ejecución en los entornos de desarrollo. Véase la sección refsec:process-declaration para obtener más información.

Tipos de procesos y procesos en ejecución

Para que una aplicación web pueda escalar horizontalmente en la plataforma *Heroku*, es necesario aprovechar la relación entre los tipos de procesos y el número de procesos en ejecución.

Un tipo de proceso es una plantilla desde la cual uno o más procesos pueden ser creados. Esto es similar al modo en que una clase sirve como prototipo para uno o más objetos instanciados en la programación orientada a objetos. En la figura 3.2 se ilustra la relación entre los procesos en ejecución de una aplicación (eje vertical) y los tipos de procesos (en el eje horizontal).

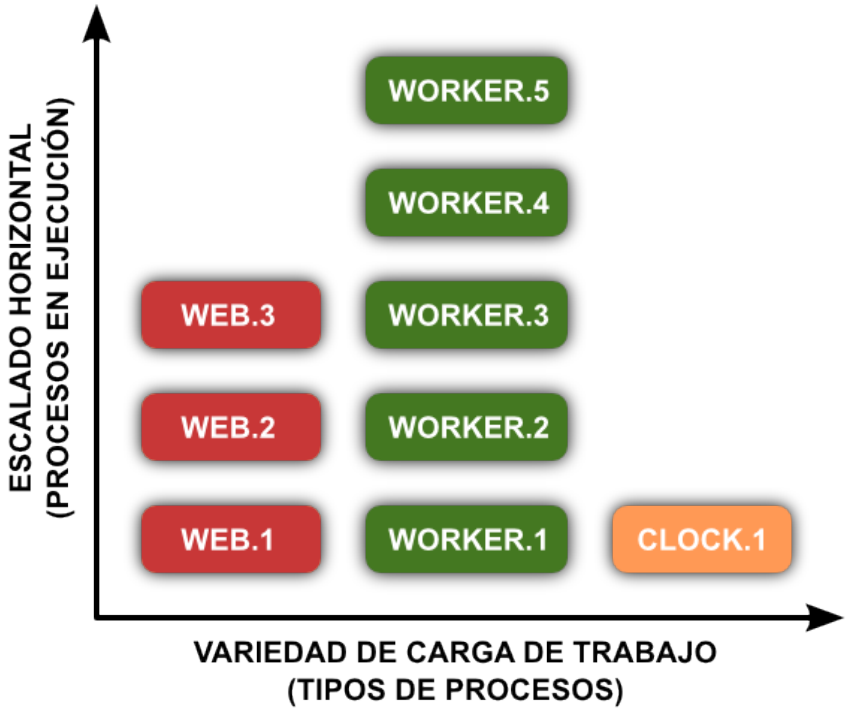

Figura 3.2: Tipos de procesos frente a procesos en ejecución

Los procesos en ejecución, en el eje vertical, permiten el escalado horizontal. Se crece a lo largo de este eje cuando es necesario incrementar la concurrencia para un tipo de trabajo determinado (tipo de proceso). En la plataforma *Heroku*, esto es posible mediante el uso del comando `heroku ps:scale` como se indica a continuación:

```
$ heroku ps:scale web=3 worker=5 clock=1
Scaling web processes... done, now running 3
Scaling worker processes... done, now running 5
Scaling clock processes... done, now running 1
```

Los tipos de procesos, en el eje horizontal, definen la diversidad de los tipos de trabajos que maneja la aplicación. Cada tipo de proceso se especializa en una determinada tarea. Véase la sección 7.5 para obtener más información sobre los tipos de procesos.

Procesos de administración

El conjunto de procesos que se ejecutan sobre el componente *dyno manifold* mediante el comando `heroku ps:scale` se denomina *process formation*. Por ejemplo: web=3; wor-

ker=5; clock=1. A parte de estos procesos que se ejecutan de forma continuada, el modelo de procesos de *Heroku* permite la ejecución de procesos puntuales (*one-off processes*) para realizar tareas administrativas como la ejecución de migraciones en la base de datos o sesiones de consola.

Procesos y *logs*

Todos los procesos que se ejecutan bajo el modelo de procesos deben escribir en la salida estándar (`STDOUT`). Cuando la aplicación se ejecuta de forma local en las máquinas de los desarrolladores, los flujos de datos escritos por los procesos son mostrados por la herramienta *Foreman* directamente en la consola de comandos. Sin embargo, en la plataforma *Heroku*, los flujos de datos de los procesos en ejecución sobre el componente *dyno manifold* son recolectados por *Logplex*[48] para ser visualizados de forma sencilla mediante el comando `heroku logs`.

3.3.4. *Dynos*

Un *dyno*, la unidad básica de composición en *Heroku*, es un contenedor ligero que ejecuta un único comando especificado por el usuario.

Características:

- ***Elasticity & scale:*** El número de *dynos* asignados a una aplicación puede ser aumentado o reducido en cualquier momento.
- ***Intelligent routing:*** La capa *Routing Mesh*, que será descrita en detalle más adelante, se encarga de ubicar todos los *dynos* que ejecutan procesos *web* y encamina el tráfico HTTP hacia ellos de forma adecuada.
- ***Process management:*** Los procesos en ejecución son controlados midiendo continuamente su capacidad de respuesta. Aquellos *dynos* cuyos procesos no funcionen correctamente son destruidos y se crean otros nuevos en su lugar.
- ***Distribution and redundancy:*** Los *dynos* están distribuidos sobre un entorno elástico de ejecución denominado *Dyno Manifold*, que será descrito en la siguiente sección. Una aplicación configurada con dos *dynos* de tipo *web* ejecuta, como cabe esperar, dos procesos *web*, pero cada uno de ellos lo hace en ubicaciones físicas distintas. Si la infraestructura subyacente fallara, la aplicación seguiría operativa.
- ***Isolation:*** Cada *dyno* está completamente aislado en su propio contenedor virtual que proporciona, entre otros beneficios, seguridad, garantía de recursos y robustez.

Memoria

Cada *dyno* tiene asignado un máximo de 512 MB de memoria. Esta configuración es válida para la mayor parte de las aplicaciones y por lo general el desarrollador no tiene que

preocuparse por ella.

En ocasiones, es posible que alguno de los procesos de un *dyno* alcance o exceda esos 512 MB de memoria asignados. Esto suele ser debido a un fenómeno conocido como fuga de memoria (*memory leak*), que sucede cuando bloques de memoria reservados por un proceso no son liberados de forma adecuada. En estos casos, lo idóneo es utilizar herramientas de análisis de uso de memoria (como *Oink*[49] para *Ruby* o *Heapy*[50] para *Python*) que ayuden a detectar el origen del problema y así poder realizar los cambios oportunos en el código fuente.

Aquellos *dynos* que exceden los 512 MB de memoria son identificados en mediante un error R14. Esto no implica que el proceso sea eliminado. Tan solo es un aviso de que el rendimiento de la aplicación se verá degradando ya que todo uso de memoria por encima de esos 512 MB será ubicado en el *swap*. Si el tamaño de la memoria en uso continua creciendo y supera 3 veces la cuota, es decir 1.5 GB, el gestor de *dynos* (*dyno manifold*) reiniciará el dyno y registra el evento con un error R15.

3.3.5. *Dyno manifold*

Se denomina *dyno manifold* al entorno de ejecución distribuido, tolerante a fallos y escalable horizontalmente para *dynos* de la plataforma *Heroku*. Es capaz de controlar casi un número infinito y diverso de programas mediante el modelo de procesos (presentado en la sección 3.3.3) de manera automática y sin mantenimiento.

Reinicios automáticos

El componente *dyno manifold* reincida todos los *dynos* de una aplicación siempre que se produzca algún cambio sobre ésta como puede ser el despliegue de nuevo código o cambios en las variables de configuración o en los *add-ons*. Es posible observar este fenómeno en tiempo real ejecutando el comando `watch heroku ps` en una consola mientras se despliega código o se alteran variables de configuración desde otra.

Los *dynos* son reiniciados al menos una vez al día o siempre y cuando el *dyno manifold* detecta un fallo en la capa de *hardware* y necesita mover los *dynos* a un nuevo emplazamiento físico. Ambas circunstancias suceden automáticamente de forma regular y son registradas en los *logs* de la aplicación.

Dynos inactivos

Aquellas aplicaciones cuyo número de *web dynos* (*dynos* que ejecutan el proceso de tipo web) en ejecución es solo uno y que no han registrado actividad por un periodo igual o

superior a una hora pasan a estado inactivo y su *dyno* deja de estar en ejecución. Cuando esto ocurre, es posible ver las siguientes líneas en el *log* de la aplicación:

```
2013-01-30T19:11:09+00:00 heroku[web.1]: Idling
2013-01-30T19:11:17+00:00 heroku[web.1]: Stopping process with SIGTERM
```

Al acceder la aplicación mediante un navegador web o cualquier otro mecanismo capaz de iniciar una petición HTTP, la malla de encaminamiento (*routing mesh*) indica al componente *dyno manifold* que despierte al *dyno* inactivo y que comience a ejecutar el proceso web:

```
2013-01-30T22:17:43+00:00 heroku[web.1]: Unidling
2013-01-30T22:17:43+00:00 heroku[web.1]: State changed from created to
starting
```

Esto provoca que la respuesta a la primera petición dure unos pocos segundos más. Las siguientes peticiones tendrán un rendimiento normal.

Las aplicaciones con más de un *web dyno* en ejecución nunca pasan a estado inactivo. Los *worker dynos* nunca pasan a estado inactivo.

Crashes

Si el proceso de un *dyno* falla (*process crash*) bien durante la fase de arranque, bien durante su operación normal, el componente *dyno manifold* inmediatamente tratará de reiniciar dicho *dyno*. Si éste falla de nuevo, se inicia una fase de reposo de 10 minutos (*ten-minute cooldown*) hasta que se intenta un nuevo reinicio. Siempre es posible realizar un reinicio manualmente mediante el comando `heroku restart`.

3.3.6. Encaminamiento de tráfico HTTP

La plataforma *Heroku* balancea y dirige automáticamente las peticiones HTTP enviadas al *hostname* de las aplicaciones hacia sus correspondientes *web dynos*. El punto de entrada para cualquier aplicación desplegada en el *stack Celadon Cedar* (entorno por defecto) es el dominio `herokuapp.com`.

Routing mesh

Las peticiones de entrada son recibidas por un balanceador de carga con soporte para los protocolos HTTP y SSL. Desde este punto pasan directamente a la malla de encaminamiento (*routing mesh*), que es el elemento encargado de identificar los *web dynos* de una aplicación dentro del componente *dyno manifold* y entregarles la petición HTTP.

La plataforma *Heroku* tiene soporte completo del protocolo HTTP 1.1 lo que permite algunas características importantes como:

- Chunked responses. La respuesta no se envía de una sola vez sino por trozos.
- Long polling. Emula notificaciones desde el servidor al cliente.
- Async webserver. Procesa múltiples respuestas simultáneamente con un solo proceso web.

Timeouts

Las peticiones HTTP disponen de una ventana de tiempo inicial de 30 segundos durante la cual el proceso web debe devolver los datos de respuesta (ya sea la respuesta completa o bien parte de la respuesta que indique que el proceso sigue activo). Los procesos que no envían datos de respuesta dentro de la ventana de tiempo inicial de 30 segundos registran un error H12 en los *logs* de la aplicación.

Después de la respuesta inicial, cada *byte* enviado (ya sea desde el cliente o desde el proceso de la aplicación) pone a cero un ventana de tiempo de 55 segundos. Si no se envían datos durante este periodo de tiempo, la conexión se cierra y se registra un error H15.

3.3.7. Aislamiento y seguridad

Tecnologías

La capa *dyno manifold* utiliza una serie de tecnologías para asegurar el aislamiento completo de los *dynos*. Algunas de las más relevantes son los siguientes:

- LXC[51] (*Linux Containers*), entorno virtual que posee su propio espacio de procesos y redes.
- Independent filesystem namespaces.
- Aislamiento del sistema de ficheros mediante la llamada de sistema `pivot_root`.

Estas tecnologías proporcionan seguridad y garantizan la asignación de recursos como tiempo de CPU y memoria en el entorno *multi-tenant* de *Heroku*.

Sistema de ficheros efímero

Cada *dyno* obtiene su propio sistema de ficheros efímero, con una copia del código fuente más reciente desplegado. Durante la vida del *dyno*, sus procesos pueden hacer uso del sistema de ficheros, pero ningún archivo modificado o creado será accesible por procesos pertenecientes a otros *dynos*. Por otro lado, cuando el *dyno* es parado o reiniciado cualquier cambio realizado en el sistema de ficheros hasta ese momento es desechado.

Direcciones IP

Cuando se utilizan varios procesos web (*web dynos*), la aplicación es distribuida sobre varios nodos del *dyno manifold*. El acceso y encaminado hacia cada *dyno* es controlado siempre por el mecanismo *routing mesh* y se realiza a través de las direcciones IP públicas de *Heroku*. Por este motivo los *dynos* no poseen direcciones IP estáticas. Además, *Heroku* controla el funcionamiento de la red y frecuentemente se mueven *dynos* entre los diferentes nodos para asegurar una optima confiabilidad y rendimiento.

3.3.8. Redundancia

Las aplicaciones que ejecutan varios *dynos* son más resistentes frente a fallos. Si algunos *dynos* dejan de estar operativos, la aplicación sigue procesado peticiones mientras los *dynos* perdidos son reemplazados por unos nuevos. Normalmente, cuando un *dyno* falla es sustituido de forma inmediata, pero en casos de fallos graves, este reemplazo puede llevar algo más de tiempo.

3.4. Conclusiones

En este tercer capítulo se ha expuesto como, en ocasiones, las soluciones *SaaS* no se ciñen a las necesidades particulares de una organización o empresa y es necesario recurrir a las plataformas en la nube (*PaaS*) sobre la cuales poder desarrollar y poner en producción una solución específica. Se han detallado además cuales con sus características esenciales, cuyo conocimiento es crucial a la hora de elegir una de ellas para un proyecto. Finalmente, se ofrece una visión detalla de la plataforma *Heroku*, seleccionada para desplegar la aplicación desarrollada en el presente trabajo.

Capítulo 4

Software as a Service

4.1. Introducción

Cualquier aplicación medianamente compleja, incluida la que ha sido implementada en el presente proyecto, necesita servicios o componentes adiciones. Algunos de estos servicios se enumeran a continuación:

- Persistencia de datos
- Almacenamiento masivo de archivos
- *Middleware* de mensajería
- Envío de SMS y/o correo electrónico
- Compresión de imágenes, vídeo y audio
- Planificador de trabajos

Tradicionalmente, cada uno de los servicios requeridos por la aplicación debían ser instalados, configurados y mantenidos por el departamento de sistemas de información. Esto, nuevamente, supone inversión en recursos como equipos *hardware*, *software*, capital humano y tiempo que finalmente se traduce en un elevado coste de operación.

Con el advenimiento de las tecnologías *cloud computing*, muchos de estos servicios han sido llevados a la nube y son ofrecidos mediante el modelo *software as a service* o *SaaS*. Este es el caso, por ejemplo, de *Heroku Postgres*[52], un servicio que permite a desarrolladores e ingenieros de *software* crear, acceder y administrar bases de datos *PostgreSQL* desde cualquier punto de la red, a la vez que les libera de engorrosas tareas como lidiar con ficheros de configuración, administrar servidores y *hardware*, instalar parches y actualizaciones, etc.

A continuación se indican qué servicios requiere la aplicación desarrollada y qué proveedores *SaaS* son los utilizados para proporcionar dichos servicios:

- **Persistencia de datos.** Los datos de cada fotografía (modelo de cámara y objetivo, tiempo de exposición, distancia focal y descripción entre otros) son almacenados en una base de datos orientada a documentos denominada *MongoDB*. El proveedor de este servicio es *MongoHQ*.

- **Almacenamiento masivo de archivos.** Los archivos JPEG de cada fotografía (así como las versiones reducidas o *thumbnails*) son almacenados en la nube utilizando el servicio *Amazon Simple Storage Service*, también conocido simplemente como *Amazon S3*.

- **Generación de diagramas.** Los gráficos circulares (diagramas de pastel) presentados en la aplicación son generados utilizando un servicio de *Google* denominado *Image Charts*.

Figura 4.1: Servicios *cloud computing* utilizados en este proyecto.

4.2. *MongoHQ*

4.2.1. Descripción

MongoHQ es una solución *SaaS* que permite crear y administrar bases de datos MongoDB[53] ubicadas en la nube. Actualmente se integra con cuatro de los mayores proveedores *PaaS* del mercado, entre los cuales se encuentra *Heroku*.

4.2.2. MongoDB

Introducción

MongoDB es una base de datos orientada a documentos de código abierto desarrollada y soportada desde 2007 por la compañía *10gen*[54]. Forma parte de una nueva familia de sistemas de almacenamiento de datos denominada *NoSQL*.

En *MongoDB*, los datos son documentos estructurados en formato *BSON* (una representación similar a *JSON*) con estructura variable y son almacenados dentro de colecciones. Cada documento contiene un elemento (`_id`) denominado *ObjectId* que actúa como clave primaria.

A continuación se muestra un sencillo documento que almacena el nombre y apellido de una persona junto con sus aficiones:

```
{
  _id: ObjectId("4be1cde88c17c20104000001"),
  name: "Alan",
  surname: "Wilson",
  hobbies: [
    "Biking",
    "Photography",
    "Programming"
  ]
}
```

La estructuración de los documentos para modelar los datos de forma efectiva es un punto crítico en *MongoDB*. Los aspectos que se deben tener en cuenta se detallan a continuación:

- *Embedding* o incrustado. Dos elementos de datos relacionados se combinan en un solo documento (denormalización). Esto proporciona dos ventajas fundamentales: rendimiento superior en operaciones de lectura y la capacidad de solicitar y recuperar los datos en una sola operación de base de datos.

- *Referencing* o referencia. Dos elementos de datos se relacionan mediante el uso de referencias (normalización). Esto evita duplicación excesiva de datos y permite la implementación de relaciones muchos a muchos y jerarquías de datos. La utilización de referencias proporciona más flexibilidad que la incrustación de documentos pero implica generalmente más interacciones con la base de datos y el rendimiento empeora.

- *Atomicity* o atomicidad. *MongoDB* solo proporciona operaciones atómicas a nivel de documento. Debido a esto, la necesidad o no de transacciones será determinante a la hora de determinar si la relación entre datos se hace mediante incrustado o referencias.

Dado que el modelo de datos de la aplicación desarrollada en el proyecto es sencillo y podría variar durante el desarrollo, el uso de *MongoDB* es aconsejable frente a otras soluciones tradicionales como *MySQL*, *Oracle* o *PostgreSQL*.

Características

Algunas características relevantes de *MongoDB* se resumen a continuación:

- Consultas específicas. *MongoDB* soporta consultas basadas en un campo, un rango o una expresión regular. El resultado puede incluir todo el documento, una parte de él e incluso incluir una función *javascript* definida por el usuario.
- Indexado. Cualquier campo de un documento en *MongoDB* puede ser indexados.
- Replicación. *MongoDB* soporta replicación maestro y esclavo. El maestro puede realizar operaciones de lectura y escritura. El esclavo copia datos desde el maestro y solo puede ser utilizado para lecturas y copias de seguridad. El esclavo tiene la posibilidad de elegir un nuevo maestro si éste deja de estar operativo por cualquier motivo.
- Balanceo de carga. *MongoDB* escala utilizando particiones de datos horizontales o *sharding*. *MongoDB* puede ejecutarse en múltiples servidores, balanceando la carga y duplicando datos para mantener el sistema siempre disponible en caso de un fallo de *hardware*. La configuración automática es de fácil despliegue y se pueden añadir máquinas nuevas en cualquier momento sin parar la base de datos.
- Almacenamiento de archivos. *MongoDB* puede utilizarse como sistema de archivos. Esta característica se denomina *GridFS*[55] y puede beneficiarse del balanceo de carga y replicación sobre múltiples máquinas para almacenar ficheros.
- Agregación. Mediante MapReduce[56] es posible realizar operaciones de agregación sobre grandes conjuntos de datos.
- *JavaScript* en el servidor. El lenguaje *JavaScript* puede ser utilizado en consultas y funciones de agregación (*MapReduce*) que son ejecutadas en el servidor.

4.2.3. Coste

El coste depende fundamentalmente de los siguientes factores:

- Base de datos dedicada o compartida

- Memoria
- Almacenamiento
- Redundancia
- Nivel de soporte

Para conocer con detalle las diferentes opciones del producto se puede es consultar su página de planes y precios[57]. Además, *MongoHQ* ofrece un plan sin coste denominado *sandbox plan*, que ofrece bases de datos compartidas, con un almacenamiento máximo de 512 MB cada una, suficiente en muchos casos y muy adecuado para realizar pruebas, pequeños proyectos, prototipos o simplemente para experimentar el funcionamiento de MongoDB. Este es el plan utilizado durante el desarrollo de la aplicación de este proyecto.

4.2.4. Herramientas

Consola de administración

MongoHQ pone a disposición de los desarrolladores una consola de administración web mediante la cual es posible:

- Crear y eliminar bases de datos
- Añadir y eliminar colecciones
- Añadir, modificar y eliminar documentos
- Añadir índices
- Copiar datos

4.3. *Amazon S3*

4.3.1. Descripción

Amazon S3[58] es un servicio ofrecido por *Amazon Web Services (AWS)* que proporciona almacenamiento masivo de archivos (denominados objetos) en Internet. Para ello, pone a disposición de los desarrolladores una interfaz de servicios web que permite almacenar y recuperar la cantidad de datos que se desee, en el instante que se desee, y desde cualquier punto de la red. *Amazon S3* es la misma infraestructura económica, altamente escalable, fiable, segura y rápida que utiliza *Amazon* para tener en funcionamiento su propia red internacional de sitios web.

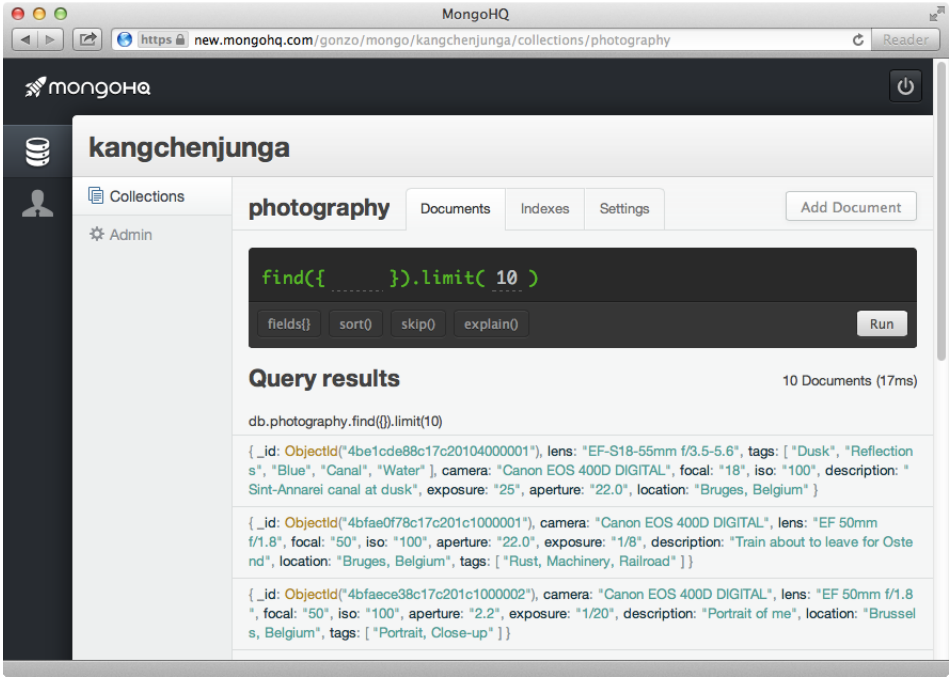

Figura 4.2: Consola de administración de *MongoHQ*

4.3.2. Funcionalidad

Amazon S3 implementa actualmente, como mínimo, la siguiente funcionalidad:

- Es posible escribir, leer y eliminar objetos que contengan desde 1 byte hasta 5 terabytes de datos. El número de objetos que se pueden almacenar es ilimitado.
- Cada objeto está almacenado en un depósito, y se recupera por medio de una clave exclusiva asignada por el desarrollador.
- Un depósito puede estar almacenado en una de varias regiones[59]. Se recomienda escoger una región cercana para optimizar la latencia, minimizar los costes o afrontar exigencias reguladoras.
- Los objetos almacenados en una región nunca abandonan la misma, a menos que se transfieran explícitamente. Por ejemplo, los objetos almacenados en la región UE (Irlanda) nunca salen de la UE.
- Se establecen mecanismos de autenticación diseñados para garantizar que los datos se mantienen seguros frente a accesos no autorizados. Los objetos pueden hacerse privados o públicos, y pueden otorgarse permisos a usuarios determinados.
- Es posible la carga y descarga segura de objetos así como su almacenamiento cifrado para una protección de datos adicional.

- Los servicios web que proporcionan acceso a *Amazon S3* implementan un interfaz REST y SOAP y están basados en estándares lo que facilitan su utilización con prácticamente cualquier tipo de herramientas de desarrollo.

- El servicio está diseñado para ser flexible y permitir añadir fácilmente protocolos o capas funcionales. El protocolo de descarga predeterminado es HTTP aunque también se proporciona el protocolo BitTorrent[60] para reducir los costes de la distribución a gran escala.

- Proporciona opciones para llevar a cabo eliminaciones de gran volumen y recurrentes. Para eliminaciones recurrentes, se pueden definir las reglas para eliminar conjuntos de objetos tras un periodo de tiempo definido con anterioridad. Para eliminaciones únicas eficientes, se pueden eliminar hasta 1000 objetos a través de una única solicitud.

- Fiabilidad respaldada por un contrato de nivel de servicio[61].

4.3.3. Características de diseño

Amazon S3 está implementado bajo las siguientes premisas:

Seguridad

Debe proporcionar una estructura que permita al usuario mantener el control absoluto de quien tiene acceso a los datos. Asimismo, los usuarios deben tener la opción de proteger de manera fácil los datos activos e inactivos.

Fiabilidad

Los datos son almacenados con una fiabilidad del 99,999999999 % y una disponibilidad del 99,99 %. No pueden existir puntos únicos de fallo. Todos los fallos deben ser tolerados o reparados por el sistema sin ningún tipo de tiempo de inactividad.

Escalabilidad

Amazon S3 se puede escalar en lo que respecta a almacenamiento, velocidad de solicitudes y usuarios para ser compatible con un número ilimitado de aplicaciones a escala web. Utiliza el escalado horizontal como ventaja: la adición de nodos al sistema aumenta, y no disminuye, su disponibilidad, velocidad, rendimiento, capacidad y robustez.

Rapidez

Amazon S3 debe tener rapidez suficiente para admitir aplicaciones de alto rendimiento. La latencia del servidor debe ser insignificante en relación con la latencia de Internet.

Los cuellos de botella de rendimiento pueden corregirse tan solo añadiendo más nodos al sistema.

Coste

Amazon S3 está creado a partir de componentes económicos de hardware comerciales. Todo hardware falla en un momento determinado, pero esto no debe afectar al sistema global. Debe ser independiente del hardware, de forma que puedan seguir obteniéndose ahorros a medida que Amazon sigue reduciendo los costes derivados de la infraestructura.

Sencillez

Amazon S3 debe conseguir dos objetivos en este apartado. Por un lado, ser una alternativa de almacenamiento altamente escalable, fiable, rápido y económico. Por otro conseguir que cualquier aplicación pueda utilizarlo desde cualquier lugar fácilmente.

4.3.4. Coste

El coste de uso del servicio, actualmente, varía por región y está determinado por los siguientes tres factores:

- Espacio de almacenamiento de datos. Por ejemplo, hasta 1 TB de almacenamiento, el coste es $0,125 por GB por mes.
- El número y tipo de peticiones HTTP realizadas. Por ejemplo, cada 1000 peticiones `PUT`, `COPY`, `POST` y `LIST` o cada 10000 peticiones `GET` tienen un coste de $0,01. Las peticiones `DELETE` no tienen coste.
- Volumen de trasferencia de datos. Por ejemplo, hasta 10 TB de transferencia de salida, el coste es $0,120 por GB por mes. La transferencia de datos de entrada no tiene coste.

Para conocer con detalle el coste que supone el uso de este servicio y otros proporcionados por *Amazon Web Services* es recomendable utilizar la utilidad *Simple Monthly Calculator*[62].

4.3.5. Herramientas

AWS Management Console

Amazon Web Services proporciona una aplicación web, figura 4.3, que permite gestionar todos sus servicios, *Amazon S3* inclusive. Una vez creada una cuenta en *AWS* y tras haber activado el servicio *S3*, es posible acceder a esta herramienta visual que permite realizar las tareas más comunes como las que se indican a continuación:

- Crear depósitos o *buckets*
- Añadir objetos a un depósito
- Acceder y copiar objetos
- Eliminar depósitos y objetos
- Administrar la seguridad de depósitos y objetos

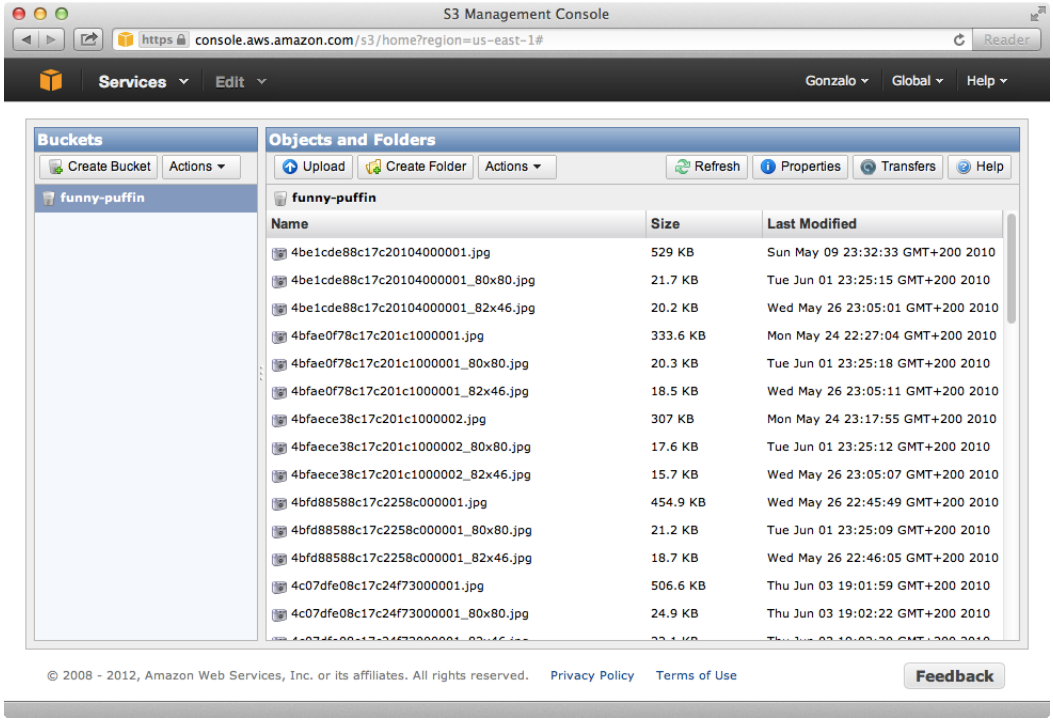

Figura 4.3: Consola de administración de *Amazon Web Services*

AWS Software Development Kit

Amazon Web Services pone a disposición de los desarrolladores bibliotecas oficiales y código fuente de ejemplo para que integrar las aplicaciones con sus servicios sea una tarea sencilla. Estas herramientas, denominadas *AWS-SDK*, se encuentran disponibles para los siguientes lenguajes y entornos de desarrollo:

- Android[63]
- iOS[64]
- Java[65]
- .NET[66]
- PHP[67]
- Ruby[68]

En la sección 6.5.2 se detalla cómo ha sido utilizada la biblioteca *AWS-SDK for Ruby* en la implementación de la aplicación.

4.4. *Google Image Charts*

4.4.1. Descripción

El servicio *Google Image Chart*[69] permite generar de forma dinámica diversos diagramas y gráficos que pueden ser incorporados en páginas web o descargados para su uso local. Su utilización no requiere la creación de una cuenta o la obtención de una clave de acceso ligada a un dominio como en el caso del servicio *Google Maps* pero está sujeta a unos términos de servicio[70].

4.4.2. Funcionamiento

El servicio *Google Image Charts* devuelve un gráfico o diagrama en forma de imagen con formato PNG como respuesta a una petición `GET` realizada contra el recurso: `https://chart.googleapis.com/chart?<params>`, donde `<params>` son las características del gráfico codificadas como parámetros de la URL. Por ejemplo, la cadena de parámetros `cht=p&chs=300x200&chd=t:20,40,40&chdl=A|B|C` produce la imagen de la figura 4.4.

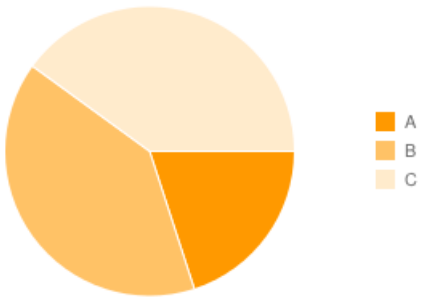

Figura 4.4: Gráfico circular generado con *Google Image Charts*.

La composición de la cadena de parámetros que define el gráfico puede llegar a ser bastante compleja por lo que es habitual ayudarse de alguna biblioteca que simplifique el proceso. La sección 6.7 detalla cómo se utiliza este servicio en la aplicación implementada en el proyecto.

4.5. Conclusiones

Las aplicaciones que se diseñan para funcionar sobre una plataforma en la nube a menudo requieren de servicios externos para suplir determinadas necesidades como la persistencia de datos. En este cuarto capítulo se presentan los servicios sobre los que depende la aplicación implementada y se ofrecen detalles sobre su funcionamiento.

Parte III

Diseño, implementación y despliegue

Capítulo 5

Diseño

5.1. Introducción

En los capítulos previos hemos presentado y explicado con cierto nivel de detalle un conjunto de tecnologías que facilitan el desarrollo y explotación de sistemas de información. Para ilustrarlo, vamos a implementar, utilizando dichas tecnologías, una sencilla aplicación similar a *Flickr* que permitirá al usuario compartir y gestionar sus fotografías.

El diagrama de casos de uso que muestra la figura 5.1 recoge los diferentes escenarios que se han contemplado.

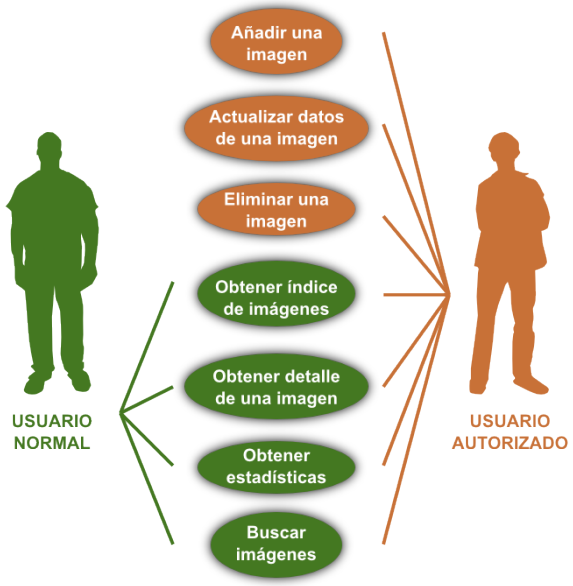

Figura 5.1: Diagrama de casos de uso

A lo largo de las siguientes secciones y con ayuda de diagramas de secuencia, se describen con detalle cada uno de los escenarios y la interacción entre los diferentes elementos del sistema.

5.2. Casos de uso

5.2.1. Añadir una imagen

Actores

Usuario autorizado.

Condiciones previas

Ninguna.

Diagrama de secuencia

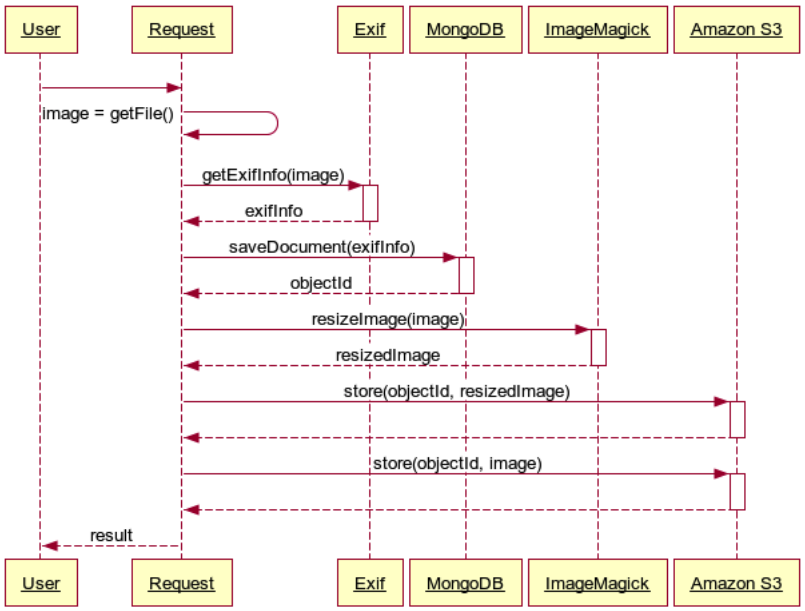

Figura 5.2: Diagrama de secuencia para añadir una imagen.

Descripción

El usuario selecciona una imagen en formato JPEG/EXIF que envía al servidor. Una vez que el servidor ha recibido la imagen, esta es procesada para obtener copias en diferentes

tamaños y extraer algunos datos como la fecha en la que fue tomada la fotografía o el modelo de cámara que fue utilizado entre otros. Las nuevas imágenes y los datos extraídos son posteriormente almacenados.

5.2.2. Actualizar datos de una imagen

Actores

Usuario autorizado.

Condiciones previas

La imagen cuyos datos se van a actualizar debe existir.

Diagrama de secuencia

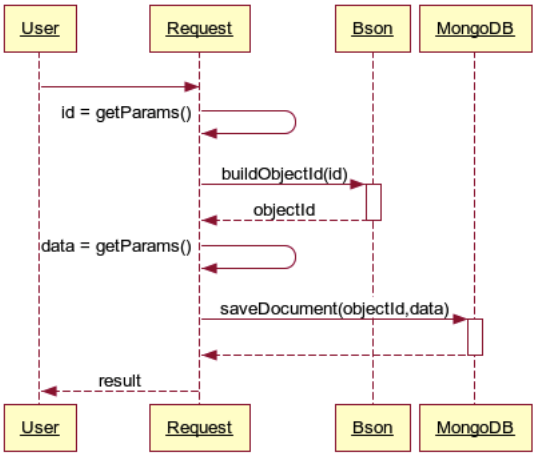

Figura 5.3: Diagrama de secuencia para actualizar datos de una imagen.

Descripción

El usuario selecciona la imagen cuyos datos desea actualizar y selecciona la acción `EDIT`. Todos los atributos para dicha imagen son presentados en un formulario que el usuario puede modificar. Una vez que los cambios se han realizado, el usuario selecciona la acción `SEND` y los datos se guardan.

5.2.3. Eliminar una imagen

Actores

Usuario autorizado.

Condiciones previas

La imagen que se va a eliminar debe existir.

Diagrama de secuencia

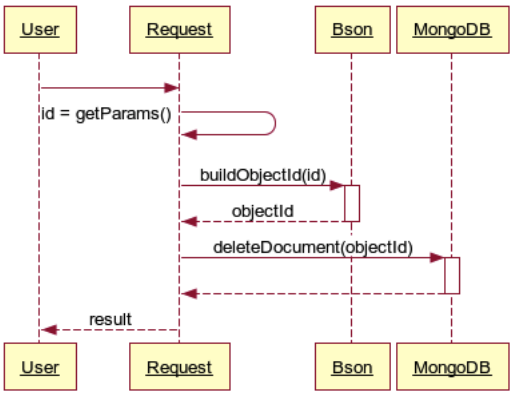

Figura 5.4: Diagrama de secuencia para eliminar una imagen.

Descripción

El usuario selecciona la vista de detalle de la imagen que quiere eliminar y utiliza la acción `DELETE`. La imagen, sus copias en menor tamaño y los datos asociados quedan eliminados del sistema tras confirmar un mensaje de alerta.

5.2.4. Obtener índice de imágenes

Actores

Usuario normal o usuario autorizado.

Condiciones previas

Ninguna.

Diagrama de secuencia

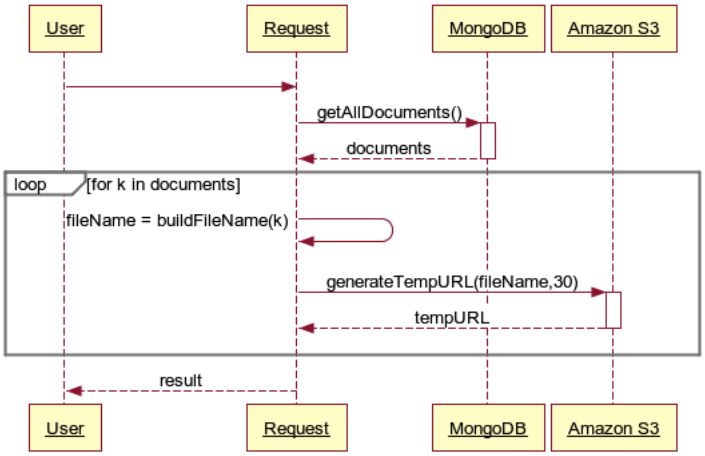

Figura 5.5: Diagrama de secuencia para obtener índice de imágenes.

Descripción

El usuario obtiene del servidor un *collage* con todas las imágenes que actualmente existen en el sistema en tamaño reducido.

5.2.5. Obtener detalle de una imagen

Actores

Usuario normal o usuario autorizado.

Condiciones previas

La imagen de la que se obtiene el detalle debe existir.

Diagrama de secuencia

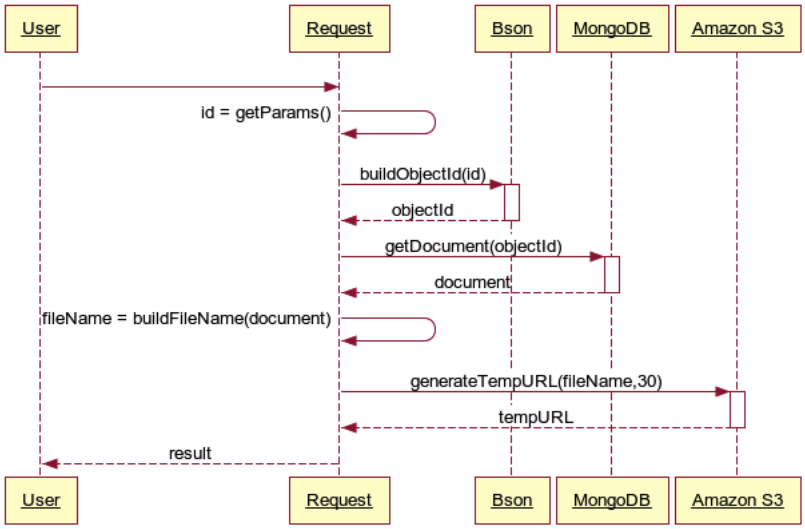

Figura 5.6: Diagrama de secuencia para obtener detalle de una imagen.

Descripción

El usuario selecciona una de las imágenes presentadas en el índice de imágenes o tras realizar una búsqueda y obtiene la imagen en tamaño completo junto a la información asociada, como por ejemplo, el modelo de cámara que se utilizó en la toma, la sensibilidad ISO, la distancia focal o el tiempo de exposición entre otros.

5.2.6. Obtener estadísticas

Actores

Usuario normal o usuario autorizado.

Condiciones previas

Deben existir imágenes en el sistema.

Diagrama de secuencia

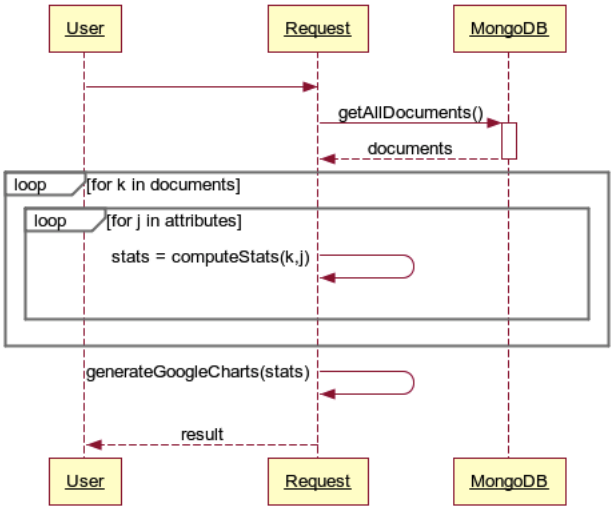

Figura 5.7: Diagrama de secuencia para obtener estadísticas.

Descripción

El usuario solicita las estadísticas de las imágenes que han sido incorporadas hasta ese momento en el sistema. Por cada atributo se muestra un gráfico indicando la distribución de valores. Por ejemplo, para el atributo *Lens* (objetivo o lente) el resultado podría ser el mostrado en la figura 5.8.

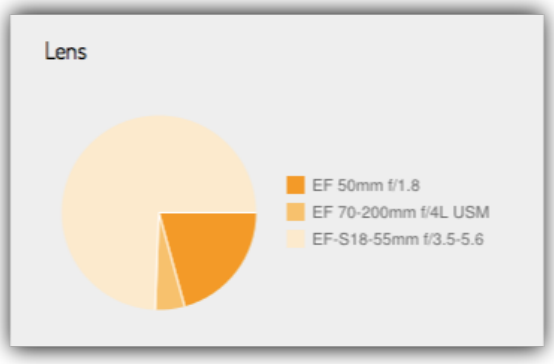

Figura 5.8: Ejemplo de visualización para el atributo *Lens*

5.2.7. Buscar imágenes

Actores

Usuario normal o usuario autorizado.

Condiciones previas

Deben existir imágenes en el sistema.

Diagrama de secuencia

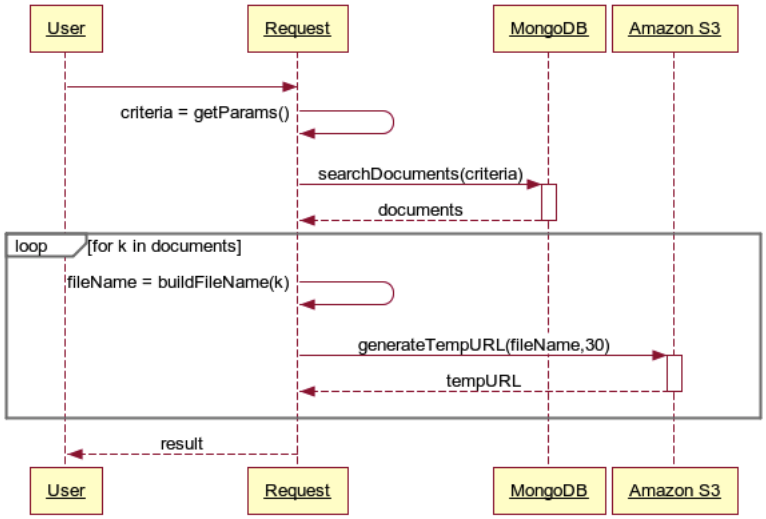

Figura 5.9: Diagrama de secuencia para buscar imágenes.

Descripción

El usuario especifica los criterios por los que quiere realizar la búsqueda, por ejemplo ISO = 800 y distancia focal = 18 (milímetros). El sistema recupera todas las imágenes cuyos atributos concuerdan con los criterios de la búsqueda y se ofrecen al usuario de forma similar a como se muestran en el caso de obtener el índice de imágenes.

5.3. Conclusiones

En este capítulo se ha presentado cual debe ser la funcionalidad básica de la aplicación. Además, mediante diagramas de secuencia se puede observar la interacción entre los diferentes elementos de la aplicación así como la interacción con los servicios auxiliares proporcionados por terceros.

Capítulo 6

Implementación

6.1. Introducción

6.1.1. *Ruby*

Ruby es un lenguaje de programación de propósito general, interpretado, orientado a objetos, de código abierto, fácil de aprender, potente y expresivo creado por Yokihiro Matsumoto. Desde que vio la luz en Japón en el año 1995 ha ganado muchos adeptos, especialmente debido a *Ruby on Rails*, un *framework* que permite escribir aplicaciones web de forma ágil. Cuenta con una extensa colección de bibliotecas (denominadas gemas) que permiten desarrollar aplicaciones en muy poco tiempo. El núcleo de *Ruby* está escrito en el lenguaje de programación C y está disponible para numerosos sistemas, incluidos *Windows*, *OS X* y *Linux*. Además, como ya se expuso en la sección 3.2.1, *Ruby* es uno de los lenguajes más soportados por los proveedores de servicios en la nube, entre los que se encuentra *Heroku*, proveedor *PaaS* sobre la cual se desplegará la aplicación.

Por todo ello, *Ruby* es el lenguaje de programación seleccionado para implementar la aplicación del presente proyecto. En la sección 7.2 se describe en detalle cómo realizar la instalación de *Ruby* y otros elementos necesarios. Para obtener más información sobre *Ruby*, es recomendable consultar la documentación que se pueden encontrar en su sitio web oficial[72].

6.1.2. *Sinatra*

Sinatra[73] es una gema que proporciona un DSL (*Domain Specific Language*) especialmente diseñado para construir aplicaciones y servicios web en *Ruby*. Su objetivo es proporcionar exclusivamente los elementos y mecanismos esenciales para lidiar con peticiones HTTP realizadas por los clientes y generar las respuestas pertinentes. A continuación se indican brevemente algunos de sus aspectos más característicos:

- Al contrario que *frameworks* como *Rails (Ruby on Rails)*[74], *Ramaze*[75] o *Merb*[76]

con un propósito similar, *Sinatra* no proporciona integración con elementos adicionales como ORM (*Object-Relational Mapper*) que permiten el acceso a una base de datos o motor de *templates* para generar vistas complejas.

- No obliga a utilizar ningún patrón de diseño en particular (como puede ser MVC) aunque sí sugiere que exista una relación cercana entre los *service endpoints* y los verbos HTTP. Esto hace a *Sinatra* muy adecuado para implementar servicios web y APIs.

- Las aplicaciones basadas en *Sinatra* pueden diseñarse para ser empotradas en otras aplicaciones (*modular mode*) o para ser aplicaciones independientes (*classic mode*). La aplicación implementada en este proyecto esta diseñada como aplicación independiente.

- Es una biblioteca bien probada, y utilizada en producción por importantes compañías como *GitHub*[77], *BBC*[78], *Engine Yard*, *Apple*[79], etc.

- Ha inspirado proyectos similares en otros lenguajes como *Express*[80] (*Node.js*), *Nancy*[81] (*Microsoft .Net*), *Mercury*[82] (*Lua*) y otros muchos por lo que su conocimiento puede ser extrapolable.

6.2. Arquitectura

Como se ha indicado en la sección anterior, *Sinatra* no obliga o fuerza a utilizar ningún patrón arquitectónico. En determinadas ocasiones, sobre todo en aplicaciones o servicios web muy sencillos, puede ser factible no utilizarlos. Sin embargo, en aplicaciones complejas a menudo se identifican problemas generales que pueden resueltos de forma efectiva utilizando algunos de ellos.

En el caso de la aplicación desarrollada en este proyecto, surge la necesidad de separar las responsabilidades de cada uno de los elementos que la componen con el objetivo de facilitar el mantenimiento y la reutilización del código. Para ello se ha tenido en cuenta las directrices del patrón Modelo-Vista-Controlador (MVC) que distingue los siguientes tres tipos de componentes y relaciones entre ellos:

- **Modelo** Representa los datos del dominio como la información EXIF de las fotografías.

- **Vista** Realiza la presentación de la información en formato HTML.

- **Controlador** Recibe las acciones de los usuarios y se comunica con el modelo y la vista para generar y entregar la respuesta.

Estos elementos se pueden identificar en cada uno de los servicios o *endpoints* que componen la aplicación como se muestra en el siguiente fragmento:

```
get '/photos' do                                                    |
  items = Photo.all                                 <--- Model      |
                                                                    |- Controller
  erb :index, :locals => { :photos => items }       <--- View       |
end                                                                 |
```

Como se puede observar en la figura 6.1, los pasos que se suceden cuando la aplicación recibe una petición (por ejemplo `GET /photos`) son los siguientes:

1. `[REQUEST]` Se determina el controlador adecuado en base al recurso solicitado y se le entrega la petición.
2. `[FETCH DATA]` El controlador solicita los datos necesarios al modelo.
3. `[UPDATE]` Con los datos recuperados, el controlador solicita a la vista la presentación.
4. `[REPONSE]` El controlador genera una respuesta donde incluye la presentación.

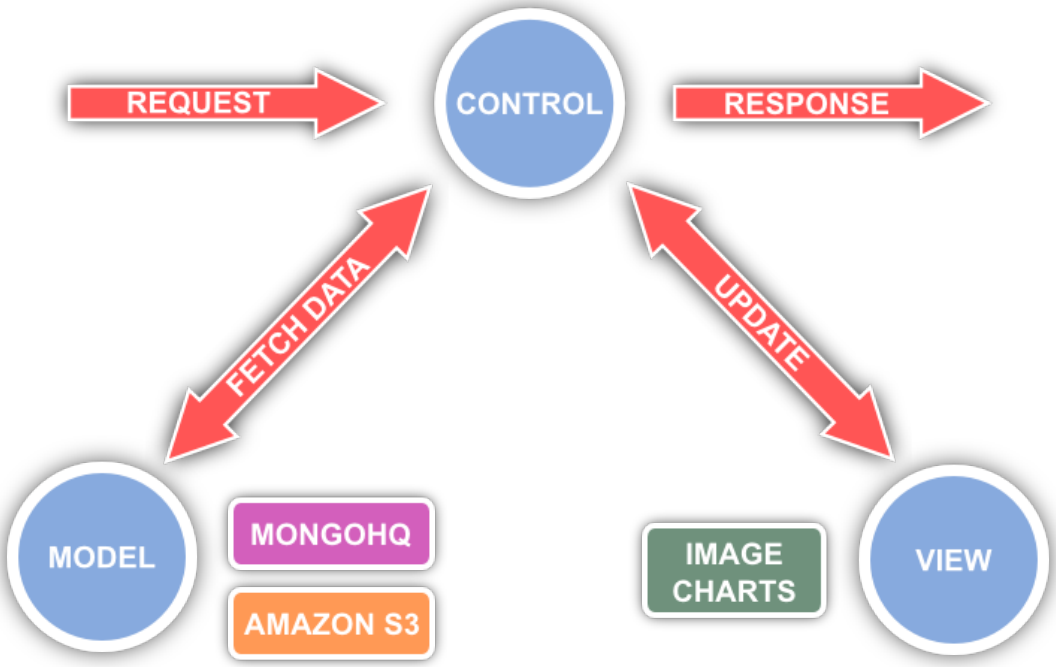

Figura 6.1: Patrón Modelo-Vista-Controlador utilizado en la aplicación del proyecto.

6.3. Estructura

6.3.1. Configuración

Sinatra define un conjunto de *settings* o configuraciones predefinidas que permiten activar y desactivar ciertas características o comportamientos de la aplicación. Estas configuraciones son simplemente variables globales de la aplicación y son modificadas mediante los métodos `set`, `enable` y `disable`. Además de las configuraciones *de fábrica*, es posible añadir configuraciones propias y todas ellas son accesibles desde la aplicación a través del objeto `settings`. Es habitual agrupar las configuraciones por entornos de ejecución como se muestra en el siguiente ejemplo:

```
configure [:development, :test] do
  set :db, 'mysql'         # set custom setting :db with value 'mysql'
  enable :lock, :sessions  # built-in settings :lock and :session enabled
  disable :raise_errors    # built-in setting :raise_errors disabled
end

configure [:production] do
  set :db, 'oracle'        # set custom setting :db with value 'oracle'
end

puts settings.db           # 'mysql' if environment is :development or
                           # :test
                           # 'oracle' if environment is :production
```

Sinatra sigue una norma de diseño de *software* denominada convención sobre configuración que busca reducir el número de decisiones que tiene que tomar un desarrollador con el fin de proporcionar una mayor simplicidad sin comprometer la flexibilidad. Por este motivo, las configuraciones predefinidas (o *built-in settings*) toman unos valores por defecto que, en la mayoría de los casos, son válidos y hace innecesario tener que definirlas explícitamente.

Sin embargo, en ocasiones puede ser necesario realizar alguna configuración especial o incorporar algún elemento *software* adicional para un entorno de ejecución determinado. En el caso de la aplicación desarrollada en este proyecto, cuando la ejecución tiene lugar en un entorno de producción (típicamente en *Heroku*), se incorpora un *middleware* que permite el control del rendimiento de la aplicación mediante la solución *APM* (Application Performance Management) de *New Relic* como se muestra en la sección `configure` de la aplicación:

```
configure :production do
  require 'newrelic_rpm'  # add New Relic middleware only in production
end
```

A continuación se describen las diferentes configuraciones predefinidas en *Sinatra* junto con sus valores por defecto que son utilizados por la aplicación:

`:environment`

Determina el entorno de ejecución. Normalmente los valores posibles que puede tomar son `:development`, `:test` o `:production`. El valor que toma por defecto es el indicado por la variable de entorno `RACK_ENV`. Si tal variable no existe, el valor que toma es `:development`.

`:sessions`

Proporciona soporte para el uso de sesiones cifradas basadas en *cookies*. Está desactivado por defecto.

`:logging`

Proporciona soporte para registrar peticiones HTTP utilizando el formato de *Apache*. Para aplicaciones *Sinatra* en modo clásico, como la implementada en este proyecto está activo por defecto. A continuación se muestra el aspecto de estos mensajes:

```
192.168.1.36 - - [26/Nov/2012 19:35:51] "GET / HTTP/1.1" 302 - 0.0017
192.168.1.36 - - [26/Nov/2012 19:35:51] "GET /photos HTTP/1.1" 200 68...
192.168.1.36 - - [26/Nov/2012 19:35:51] "GET /css/reset.css HTTP/1.1"...
192.168.1.36 - - [26/Nov/2012 19:35:51] "GET /css/style.css HTTP/1.1"...
192.168.1.36 - - [26/Nov/2012 19:35:51] "GET /assets/upload.png HTTP/...
[...]
```

`:method_override`

Controla el uso del parámetro `_method` en las peticiones HTTP POST. Cuando está activo (valor por defecto) el método HTTP utilizado (`POST`) es sustituido por el indicado en el parámetro `_method` incluido en el cuerpo de la petición. Este mecanismo permite hacer pasar peticiones `POST` por otro tipo de peticiones como `PUT` o `DELETE`. Habitualmente, este *truco* es utilizado en situaciones donde no está soportado el conjunto completo de verbos o métodos HTTP, como es el caso del envío de un formulario HTML desde un navegador web. A continuación se muestra un ejemplo:

```
<form name="form" action="/users/23" method="post" enctype="multipart...">
<input type="hidden" name="_method" value="delete"/>
[...]
</form>
```

`:app_file`

Determina el lugar donde reside el fichero principal de la aplicación. Se utiliza para establecer el *path* o ruta por defecto para directorios importantes como los definidos por las configuraciones `:root`, `:public_folder` y `:views`. *Sinatra* utiliza una heurística para determinar el valor por defecto para esta configuración, pero se puede especificar de forma explícita de la siguiente manera:

```
set :app_file, __FILE__  # __FILE__ in Ruby is a magic variable that
                         # contains the name of the current file
```

`:root`

Determina el directorio base o raíz de la aplicación. Por defecto, se asume que el directorio raíz es aquel en el que reside el fichero principal de la aplicación (`:app_file`). Este directorio es empleado para construir otros directorios relevantes como los definidos por las configuraciones `:public_folder` y `:views`. Se puede definir de forma explícita de la siguiente forma:

```
set :root, File.dirname(__FILE__)
```

`:static`

Determina si se deben servir recursos estáticos desde el directorio público de la aplicación (`:public_folder`). Cuando esta configuración está activa (valor por defecto), *Sinatra* comprueba si debe servir un recurso estático antes de buscar coincidencias en las rutas definidas.

`:public_folder`

Determina el directorio desde el que se sirve contenido estático. Por defecto, se asume que el nombre de este directorio es `public` y que se encuentra en el directorio raíz de la aplicación. Se puede especificar de forma explícita de la siguiente forma:

```
set :public_folder, '/var/share/assets'
```

La manera recomendada de especificar un directorio relativo a la configuración `:root` es la siguiente:

```
set :public_folder, Proc.new { File.join(root, 'static') }
```

`:views`

Determina el directorio en el que se ubican las plantillas para las vistas. Por defecto, se asume que el nombre de este directorio es `views`, y que se encuentra en el directorio raíz de la aplicación. Es posible especificar un nuevo directorio de la siguiente forma:

```
set :views, Proc.new { File.join(root, 'html') }
```

`:run`

Determina si *Sinatra* debe iniciar un servidor web una vez la aplicación ha sido cargada. Por defecto esta configuración está activa siempre que el fichero de la aplicación (`:app_file`) coincida con la variable `$0`. Esto ocurre cuando se arranca la aplicación directamente utilizando el interprete `ruby`. Por ejemplo:

```
$ ruby1.9 app.rb
```

`:server`

Define una lista de servidores web compatibles con *Rack*. Cuando la configuración `:run` está activa, *Sinatra* recorre esta lista buscando el primer servidor web disponible para arrancarlo. El valor por defecto está definido de la siguiente forma:

```
set :server, %w[thin mongrel webrick]
```

`:bind`

Determina el nombre de máquina o dirección IP del interfaz de red que se pondrá en escucha si la configuración `:run` está activada. El valor por defecto es `0.0.0.0` que significa que escucha en todos los interfaces. Para que solo escuche en el interfaz local o *loopback*:

```
set :bind, 'localhost'
```

`:port`

Determina el puerto que se utiliza cuando la configuración `:run` está activada. Por defecto toma el valor `4567`. También se puede especificar cuando se inicia la aplicación de la siguiente manera:

```
$ ruby1.9 app.rb -p 5000  # start on port 5000
```

`:dump_errors`

Determina si los *backtraces* son escritos a la salida estándar de error (`STDERR`) cuando una excepción no controlada sucede desde una ruta o un filtro. Por defecto está activado en aplicaciones *Sinatra* en modo clásico.

`:raise_errors`

Determina si la excepciones ocurridas dentro de rutas o filtros deben tener alcance fuera de la aplicación. Si está desactivado, las excepciones son capturadas para generar una respuesta con código `5XX` y se muestra una página de error. Si está activado los errores se propagan fuera de la aplicación lo que puede producir su parada. Por defecto solo está activo cuando el entorno de ejecución es `:test`.

`:default_encoding`

Determina el tipo codificación. Por defecto es `utf-8`.

`:lock`

Sinatra está diseñado para funcionar en un entorno con varios hilos de ejecución permitiendo el procesado de más de una petición al mismo tiempo. Sin embargo, no todas las gemas que se pueden utilizar en una aplicación *Sinatra* son *thread safe* lo cual podría provocar fallos intermitentes y comportamientos erráticos. Al activar `:lock`, todas las peticiones son sincronizadas mediante un mecanismo de exclusión mutua (*mutex*) asegurando que solo una petición es atendida a la vez. Este ajuste está desactivado por defecto.

`:show_exceptions`

Permite que se muestren páginas de error con información relevante cuando se produce una excepción no controlada. Por defecto está activado si la variable global `:environment` tiene el valor `:development`.

6.3.2. Rutas y recursos

El propósito fundamental de cualquier aplicación *Sinatra* es poder responder a una serie de rutas. Las rutas son el mecanismo principal mediante el cual los usuarios u otros sistemas se comunican con la aplicación.

Este proceso de comunicación está sustentado en el intercambio de mensajes HTTP. Cada uno de ellos está compuesto por las siguientes secciones:

- ***Request line***

  Define el método HTTP que se va a utilizar, así como el recurso (ruta) que se solicita y la versión del protocolo HTTP necesario para que el servidor determine cómo analizar el mensaje. Por ejemplo: `GET /photos HTTP/1.1`

- ***Headers***

  Proporcionan información adicional. La especificación HTTP define una serie de cabeceras (*headers*) estándar que cubren la mayor parte de las necesidades aunque es posible incorporar nuevas personalizadas. Cada cabecera posee un nombre y un valor separados por el signo de puntuación `:` (dos puntos) y está ubicada en una línea. En el siguiente ejemplo se muestran tres cabeceras:

  ```
  User-Agent: curl/7.28.1
  Host: localhost
  Accept: */*
  ```

- ***Body***

  Por último la sección *body* es el cuerpo del mensaje y puede contener cualquier tipo de contenido, tanto binario (imágenes, audio...) como texto (HTML, XML...).

Por otro lado, HTTP define una serie de métodos o verbos que cualifican una petición. A continuación se indican los más importantes y su cometido:

- `GET` Solicita a la aplicación la representación de un recurso.
- `POST` Envia datos a la aplicación.
- `PUT` Crear o actualiza (sustituye completamente) la representación de un recurso de la aplicación.
- `DELETE` Eliminar un recurso de la aplicación.
- `PATCH` Actualiza parcialmente un recurso de la aplicación.

Por cada uno de los verbos definidos en el estándar HTTP, *Sinatra* proporciona un método para construir un servicio o *endpoint*. Estos métodos reciben como parámetros una o varias rutas (*routes*), una serie de condiciones (opcional) y un bloque de código que define la lógica de negocio del *endpoint* y que proporciona la respuesta.

A continuación se muestra la definición general y un ejemplo sencillo:

```
<http_method> <routes>, [conditions] do
  # business logic block
end

get ['/time', '/hora'] do
  Time.now.to_s
end
```

En ejemplo anterior se define un recurso accesible mediante el método `GET` desde dos rutas, `/time` y `/hora` cuya respuesta el la hora del sistema. Las rutas son evaluadas en el orden en que han sido definidas y la primera que coincide es la que recibe el control.

La aplicación desarrollada distingue tres tipos de recursos: las fotografías, las búsquedas y las estadísticas. Siguiendo la filosofía de diseño *REST* (*Representational State Transfer*)[83, Capítulo 4] se han definido las rutas recogidas en la tabla 6.1.

Método	Ruta	Uso
`GET`	`/photos`	Muestra el índice con todas las fotos.
`GET`	`/photos/new`	Formulario HTML para añadir una foto nueva.
`POST`	`/photos`	Añade una foto nueva.
`GET`	`/photos/:id`	Muestra el detalle de una foto.
`GET`	`/photos/:id/edit`	Formulario HTML para editar datos de una foto.
`PUT`	`/photos/:id`	Actualiza los datos de una foto.
`GET`	`/photos/:id/remove`	Formulario HTML para borrar una foto.
`DELETE`	`/photos/:id`	Elimina una foto.
`GET`	`/search/photos`	Formulario HTML para realizar un búsqueda.
`GET`	`/search/photos/:criteria`	Muestra el resultado de la búsqueda.
`GET`	`/stats/photos`	Muestra gráficos con estadísticas.

Tabla 6.1: Rutas de acceso a los recursos de la aplicación del proyecto

6.4. *Helpers*

Los *helpers* son métodos auxiliares cuyo objetivo es encapsular funcionalidad que pueda ser utilizada en varios puntos de la aplicación, fundamentalmente desde la definición de las rutas (*handlers*) y las vistas.

6.4.1. *Built-in*

Sinatra proporciona una serie de *helpers* por defecto que permiten acelerar y simplificar el desarrollo. A continuación se enumeran y explican algunos de los utilizados para la implementación del proyecto.

`halt`

Interrumpe inmediatamente el procesamiento de la petición HTTP en curso. Este método puede recibir diversos parámetros como se muestra a continuación:

```
halt
halt 400                       # status code
halt 'response body'           # response content
halt 400, 'response body'      # status code + response content
halt 400, {}, 'response body' # status code + headers + response content
halt erb(:error)               # render view
```

`redirect`

Permite dirigir al cliente (habitualmente un navegador web) hacia otro recurso. A continuación se muestra un ejemplo extraído de la aplicación:

```
get '/' do
  redirect to '/photos'
end
```

En este caso, cuando el cliente solicite el recurso `/`, la aplicación generará una respuesta con código de estado `"HTTP/1.1 302 Moved Temporarily"` y la cabecera `"Location: http://<domain>/photos"`.

Es importante recalcar que en este ejemplo, el método `redirect` toma como parámetro el resultado de la invocación de `to '/photos'`. `to` es un *helper* cuyo significado se detalla más adelante.

`status`

Determina el código de estado de la respuesta. A continuación se muestra un ejemplo extraído de la aplicación:

```
get '/photos/:id' do
  id = params[:id]
  item = Photo.find(id)
```

```
  if item.nil?
    status 404
  else
    [...]
  end
end
```

En este caso, si el identificador proporcionado en la petición no corresponde con ninguna fotografía, el código de estado de la respuesta será `"HTTP/1.1 404 Not found"`.

`url, to`

El método `url` y su alias `to` permiten generar URLs.

`request`

El objeto `request` representa la petición de un cliente. Los siguientes métodos (solo se enumeran los más relevantes) permiten obtener información sobre ella:

- `request.accept`: tipo de contenido que acepta el cliente. Por ejemplo: `['text/html', */*']`.
- `request.body`: contenido del cuerpo (*body*) de la petición.
- `request.scheme`: protocolo utilizado por la petición. Por ejemplo: `"http"`.
- `request.port`: puerto utilizado por la petición. Por ejemplo: `"80"`.
- `request.request_method`: método HTTP empleado por el cliente. Puede tomar los valores: `"GET"`, `"POST"`, `"PUT"`, `"DELETE"` y `"PATCH"`.
- `request.content_length`: tamaño en *bytes* del cuerpo de la petición.
- `request["param_name"]`: obtención del valor del parámetro `param_name` de la petición.
- `request.env`: acceso a la estructura *hash* proporcionada por el interfaz web *Rack* con toda la información sobre la petición. Por ejemplo: `request.env['REQUEST_URI']` ofrece la misma información que el *helper* `request.path`.

6.4.2. Personalizados

Es posible añadir métodos auxiliares (*helpers*) adicionales a los definidos por defecto en *Sinatra*. Para ello se utiliza el método especial `helpers` de la siguiente manera:

```
helpers do
  # helper methods go here
end
```

Al igual que los métodos auxiliares predefinidos, estos pueden ser invocados desde rutas y plantillas. A continuación se describen con detalle los implementados en este proyecto:

`protected!`

Como ya se expuso en la sección 5.2, cierta funcionalidad como la creación, edición y borrado de fotografías, solo está disponible para usuarios autorizados. En la aplicación desarrollada, se considera que un usuario está autorizado si se ha autenticado correctamente. El mecanismo de autenticación empleado es *HTTP Authentication Basic*[84], el más sencillo de los definidos en el estándar HTTP. Utiliza las cabeceras `WWW-Authenticate` y `Authorization` para solicitar y proporcionar las credenciales respectivamente. La información intercambiada no está protegida por lo que es necesario establecer primero un canal seguro entre el cliente y el servidor mediante TLS/SSL y así evitar que ésta pueda ser interceptada y utilizada de forma fraudulenta.

A continuación se muestra la implementación del método `protected!` que gestiona la autorización/autenticación en la aplicación:

```
  def protected!
    # Force https scheme when production
    redirect
    "https://#{request.env['HTTP_HOST']}#{request.env['REQUEST_URI']}"
    if production? && !request.secure?

    auth = Rack::Auth::Basic::Request.new(request.env)

    # Request a username/password if the user does not send one
    unless auth.provided?
      response['WWW-Authenticate'] = %q{Basic realm=""}
      halt 401
    end

    # A request with non-basic auth is a bad request
    unless auth.basic?
      halt 400
    end

    # Authentication is well-formed, check the credentials
```

```
    if auth.provided? && CREDENTIALS == [auth.credentials[0],
                                         auth.credentials[1]]
      return true
    else
      halt 403
    end
  end
```

Este método realiza varias comprobaciones y acciones que se pueden pueden resumir en los siguientes cuatro pasos:

1. Si la petición HTTP se realiza a través de un canal inseguro y el entorno de ejecución es producción, el cliente es instruido para que realice la consulta utilizando el protocolo SSL/TLS. En otro caso sigue hacia el paso siguiente.

2. Si la petición HTTP no contiene información de autenticación se añade una cabecera `WWW-Authenticate` a la respuesta para solicitar al cliente que proporcione las credenciales y se para la ejecución con un error `401 Unauthorized`. En otro caso sigue hacia el paso siguiente.

3. Si la petición HTTP contiene información de autenticación para un mecanismo distinto de *HTTP Authentication Basic*, se para la ejecución con un error `400 Bad Request`. En otro caso sigue hacia el paso siguiente.

4. Si las credenciales proporcionadas en la petición HTTP no coinciden con las almacenadas en la constante de la aplicación `CREDENTIALS` la ejecución se para con un error `403 Forbidden`. En otro caso el método devuelve `true` lo que significa que al autorización/autenticación ha tenido éxito.

En la aplicación, todos los recursos que permiten la creación, modificación y borrado de fotografías invocan el método `protected!` como primer paso. A continuación se muestra un ejemplo extraído del código:

```
get '/photos/new' do
  protected!
  erb :new
end
```

`link`

Por otro lado, ha sido necesario crear un pequeño método auxiliar llamado `link` que permite simplificar la creación de enlaces al recurso `/search/photos` como se muestra en el siguiente extracto de la plantilla `detail.erb`:

```
Lens: <a href=<%=to(link(photo, :lens))%>><%=photo[:lens]%></a>

# code above will render to:
#
Lens: <a href=/search/photos?lens=EF+50mm+f/1.8>EF 50mm f/1.8</a>
```

La llamada al método `link` se realiza con dos argumentos. El primero es la variable local `photo`, una estructura de datos *hash* que contiene los atributos de la fotografía. El segundo es la clave o atributo para el cual se quiere generar el enlace. A continuación se muestra la implementación y un ejemplo:

```
  def link(h, k)
    "search/photos?#{k}=#{h[k].split.join('+')}"
  end

  # example
  #
  hash = {:iso => '100', :lens => 'EF 50mm f/1.8', [...]}
  link(hash, :lens)  # 'search/photos?lens=EF+50mm+f/1.8'
```

6.5. Persistencia

6.5.1. Datos EXIF

Los datos EXIF de cada fotografía son almacenados en una base de datos orientada a documentos llamada *MongoDB*. Cada fotografía está representada como un documento JSON/BSON con una estructura similar a la que se muestra en el siguiente ejemplo:

```
{
  _id: ObjectId("4be1cde88c17c20104000001"),
  lens: "EF-S18-55mm f/3.5-5.6",
  tags: [
    "Dusk",
    "Reflections",
    "Blue",
    "Canal",
    "Water"
  ],
  camera: "Canon EOS 400D DIGITAL",
  focal: "18",
  iso: "100",
```

```
  description: "Sint-Annarei canal at dusk",
  exposure: "25",
  aperture: "22.0",
  location: "Bruges, Belgium"
}
```

Estos documentos se almacenan en una estructura superior denominada colección. El nombre de la colección es `photography`.

Para acceder a la información de la base de datos se utiliza un ORM denominado *MongoMapper*[85]. Existen otros muchos, algunos de los cuales son *Mondoid*[86], *MongoODM*[87] y *MongoModel*[88].

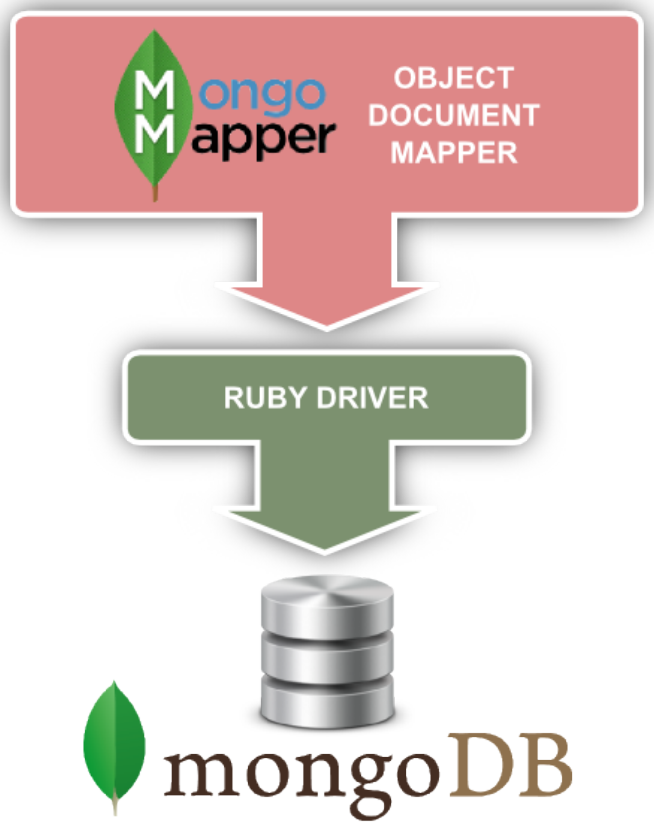

Figura 6.2: *Object Document Mapper*

En el siguiente cuadro se muestra la sección de la aplicación donde realiza la configuración para utilizar este servicio:

```
MONGOHQ_URL = 'mongodb://<user>:<pass>@flame.mongohq.com:27084/<db>'
MongoMapper.config = { 'environment' => {'uri' => MONGOHQ_URL}}
MongoMapper.connect('environment')
```

La información de cada fotografía (distancia focal, modelo de cámara y objetivo, apertura entre otros) se ha modelado mediante una clase *Ruby* denominada *Photo*:

```
class Photo
  include MongoMapper::Document

  safe

  key :camera, String
  key :lens, String
  key :focal, String
  key :iso, String
  key :aperture, String
  key :exposure, String
  key :description, String
  key :location, String
  key :tags, Array

  attr_accessible :camera, :lens, :focal, :iso,
                  :aperture, :exposure,
                  :description, :location, :tags

  set_collection_name 'photography'
end
```

Dicha clase adquiere el comportamiento definido por el módulo `MongoMapper::Document` de la biblioteca *MongoMapper*. Esto, en esencia, permite que la clase `Photo` pueda ser persistida como un documento en una base de datos *MongoDB*.

Dentro de la clase se invocan algunos métodos cuyo significado se explica a continuación:

- `safe`: indica que todas las operaciones de escritura y actualización del documento `Photo` se realizan en modo seguro (la base de datos indica el resultado de la acción tras la ejecución).
- `key`: determina las claves del documento. Estas claves corresponden fundamentalmente los atributos obtenidos examinado la sección EXIF de las fotografías.

- `attr_accesible`: determina que atributos son válidos cuando se utiliza la asignación masiva.
- `set_collection_name`: permite definir el nombre de la colección sobre la que se persisten los documentos.

Adicionalmente, la clase `Photo` define los métodos `map` y `reduce` que contienen código *JavaScript* necesario para la operación *MapReduce* que tiene lugar cuando se ejecuta el recurso `/stats/photos`:

```
# Map function
#
def self.map(key)
  <<-MAP
    function() { emit(this.#{key}, 1) };
  MAP
end

# Reduce function
#
def self.reduce
  <<-REDUCE
    function(key, values) { var sum = 0;
                            values.forEach(function(doc) { sum += 1; });
                            return sum; };
  REDUCE
end
```

La función *MapReduce* permite obtener la distribución de valores para un atributo dado sobre todos los documentos almacenados en la colección `photography`. Por ejemplo, dado el atributo `:iso`, la función podría generar los siguientes documentos:

```
{ _id: "100", value: 3 }  # 3 photos ISO 100
{ _id: "200", value: 4 }  # 4 photos ISO 200
{ _id: "800", value: 1 }  # 1 photo ISO 800
```

Las operaciones fundamentales que *MongoMapper* proporciona para interactuar con la base de datos y que se han empleado en este proyecto son las siguientes:

- `Photo.all`: método de clase que permite acceder a la colección `photography` y obtener todos los documentos.
- `Photo.find(:id)`: método de clase que permite recuperar el documento de la colección `photography` cuyo *ObjectId* es igual al parámetro `:id`.

- `Photo.new(:attributes)`: método de clase que permite crear un documento cuyo contenido es el proporcionado por el parámetro `:attributes` en la colección `photography`.
- `Photo.where(:attributes)`: método de clase que permite obtener todos aquellos documentos de la colección `photography` que posean los mismos atributos y valores que los proporcionados en el parámetro `:attributes`.
- `save`: método de instancia que permite almacenar los cambios de un documento (instancia).

6.5.2. Archivos JPEG

Amazon S3 es la solución elegida para el almacenamiento masivo de fotografías. Como ya se mencionó en la sección 4.3.5, *Amazon* proporciona la biblioteca *AWS-SDK for Ruby* (gema `aws-sdk`) que permite integrar de forma sencilla aplicaciones escritas en *Ruby* con un gran número de sus servicios entre los cuales figuran *DynamoDB*, *EC2*, *CloudWatch*, *Route53* y por supuesto, *S3*.

En el siguiente cuadro se muestra la sección de la aplicación donde realiza la configuración para utilizar este servicio:

```
AMAZON_ACCESS_KEY_ID = 'RHHSVDFSSDKGQADFGHZX'
AMAZON_SECRET_ACCESS_KEY = 'zGFcv42bNnZsrr2eDawHr4NbiNQA4fx+KJaLUOi'
PHOTO_BUCKET = 'bucket-name'
bucket = AWS::S3.new(:access_key_id => AMAZON_ACCESS_KEY_ID,
                     :secret_access_key => AMAZON_SECRET_ACCESS_KEY)
                     .buckets[PHOTO_BUCKET]
```

Las primeras dos líneas definen las credenciales de acceso:

- `AMAZON_ACCESS_KEY_ID`

  Esta clave identifica a un tercero como responsable de las peticiones. Se incluye en cada una de las solicitudes al servicio y no es secreta.

- `AMAZON_SECRET_ACCESS_KEY`

  Cada clave de acceso tiene una clave secreta de acceso asociada. Esta clave es simplemente una cadena larga de caracteres que se utiliza para calcular un firma que se incluye con cada petición. Cuando la petición es recibida por AWS se comprueba la autenticidad de ésta calculando la firma utilizando la clave secreta de acceso asociada a la clave de acceso indicada en la petición. La clave secreta de acceso, como su nombre indica es secreta y solo debe ser conocida por la aplicación (desarrollador) y por AWS.

La tercera línea define una constante con el nombre del *bucket* que almacena las fotografías. La cuarta crea la instancia `bucket` mediante la cual es posible realizar las siguientes acciones:

- Almacenar una fotografía utilizando una sola petición.

```
bucket.objects["<file_name>"].write Pathname.new(file.path),
                                    :single_request => true
```

- Eliminar una fotografía.

```
bucket.objects["<file_name>"].delete
```

- Obtener la URL no segura de una fotografía con una caducidad de 30 segundos. Esto previene el fenómeno denominado *hotlinking*.

```
bucket.objects["<file_name>"].url_for(:get, :secure => false,
                                            :expires => 30)
```

6.6. Procesamiento de imágenes

6.6.1. Obtención de datos EXIF

El estándar EXIF (*Exchangeable Image File Format*, soportado por la mayoría de fabricantes de cámaras digitales, define un formato común que permite el almacenamiento de metadatos en cada fotografía. Estos metadatos pueden incluir entre otros la fecha, las coordenadas del lugar o la configuración de la cámara en el instante de la toma de la fotografía.

Para extraer esta información, la aplicación hace uso de la gema `exifr`. A continuación se muestra un extracto del código de la aplicación que muestra su uso:

```
exif = EXIFR::JPEG.new(<path_to_file>)  # extract info into exif object
exif.model                              # "Canon PowerShot G3"
exif.f_number.to_f                      # 2.2
```

6.6.2. Cambio de tamaño

Las imágenes se almacenan en tres tamaños distintos:

- **Original**

  Normalmente `900px` ancho x `<900px` alto (formato horizontal) o `<650px` ancho x `650px` alto (formato vertical). No se utiliza ningún sufijo. Ejemplo: `0001.jpg`

- **Reducido rectangular**

  `82px` ancho x `46px` alto. Se utiliza el sufijo `_82x46`. Ejemplo: `0001_82x46.jpg`

- **Reducido vertical**

  `80px` ancho x `80px` alto. Se utiliza el sufijo `_80x80`. Ejemplo: `0001_80x80.jpg`

Para realizar la conversión de tamaño se utiliza la gema `rmagick`. Esta gema simplemente proporciona un interfaz *Ruby* sobre la biblioteca *ImageMagick* escrita en lenguaje C. Por este motivo, el proceso de instalación de la gema `rmagick` require que *ImageMagick* ya esté presente. En sistemas con gestor de paquetes, la instalación de *ImageMagick* suele ser sencilla:

```
$ sudo port install imagemagick  # Mac OS X + Macports
```

La instalación de `rmagick` y el resto de gemas se detalla en el siguiente capítulo. El cambio de tamaño se realiza mediante el método `crop_resized` como se detalla en el siguiente fragmento de código:

```
Magick::Image.read(file.path)[0].crop_resized(82, 46)  # 82x46 thumbnail
Magick::Image.read(file.path)[0].crop_resized(80, 80)  # 80x80 thumbnail
```

6.7. *Image Charts*

Image Charts, como ya se expuso en la sección 4.4, es un servicio web ofrecido por *Google* que permite generar diversos tipos de diagramas (circular, barras y mapas entre otros) como imágenes en formato PNG. Para facilitar el acceso a este recurso, la aplicación hace uso de la gema `gchart`. Esta biblioteca proporciona una clase llamada `Gchart` cuyos métodos de clase permiten crear los diferentes tipos de diagramas. Estos métodos reciben como parámetros, entre otros, los datos y las leyendas que se quieren mostrar. A continuación se muestra un ejemplo de su uso en la aplicación:

```
Gchart.pie(:data => data, :legend => legend, :background => 'EEEEEE')
```

El método utilizado es `pie`, que como es lógico genera un diagrama circular. Los parámetros `data` y `legend` son vectores que contienen datos numéricos y cadenas de texto respectivamente. El último parámetro determina el color del fondo del diagrama en notación hexadecimal. El resultado de la ejecución del método es la URL del diagrama que luego es utilizada en la vista HTML.

6.8. Vistas

Las vistas son documentos HTML generados mediante un *template engine* o motor de plantillas. El objetivo principal de una plantilla es separar el contenido que se desea mostrar de su presentación final. *Sinatra* ofrece integración con varios motores como *Haml*, *Erb*, *Builder*, *Nokogiri*, *Liquid* y otros muchos. El mecanismo general para hacer uso de ellos se expone a continuación:

```
<engine_method> <template_name>, <options>

# example from code
#
erb :index, :locals => { :photos => :items }
```

`<engine_method>`

Cada motor de plantillas se expone a través de su propio método. En el ejemplo se utiliza el método `erb` ya que *Erb* es el motor de plantillas empleado en la aplicación.

`<template_name>`

Es el primer parámetro y determina qué plantilla se quiere generar. En el ejemplo se pasa el valor `:index` para indicar que se debe utilizar la vista definida en el archivo `index.erb`.

`<options>`

Es el segundo parámetro y permite especificar una serie de opciones, algunas de las más importantes se muestran a continuación:

- `:locals`: es el mecanismo mediante el cual es posible pasar datos para su presentación a las plantillas. En el ejemplo, el contiendo de la variable `items` es accesible desde la plantilla a través de la variable `photos`.
- `:default_encoding`: codificación del texto. En la sección 6.3.1 se indica el valor por defecto y como alterarlo.
- `:views`: determina el directorio desde el que cargar la plantilla. En la sección 6.3.1 se indica el valor por defecto y como alterarlo.
- `:layout`: determina si se debe utilizar un *layout* o no. Por defecto, se utiliza `layout.erb` ubicado en el directorio `:views`.
- `:content_type`: determina el tipo de contenido que se desea generar. Su valor por defecto depende del motor. Generalmente es `text/html`.

Las plantillas pueden ser de dos tipos: *layouts* o *partials*. Las primeras definen la estructura general de la vista. Las segundas presentan información particular o específica de algún elemento del dominio. El proceso de construcción de una vista, como se muestra en la figura 6.3, consta de varios pasos:

1. La plantilla de tipo *partial*, por ejemplo `:index`, recibe un objeto `:locals` con la información que se desea mostrar.

2. El motor evalúa dicha plantilla y produce un documento HTML con la estructura e información requerida.

3. Finalmente, el motor evalúa la plantilla de tipo *layout* y combina el resultado con el del paso anterior produciendo el documento HTML final.

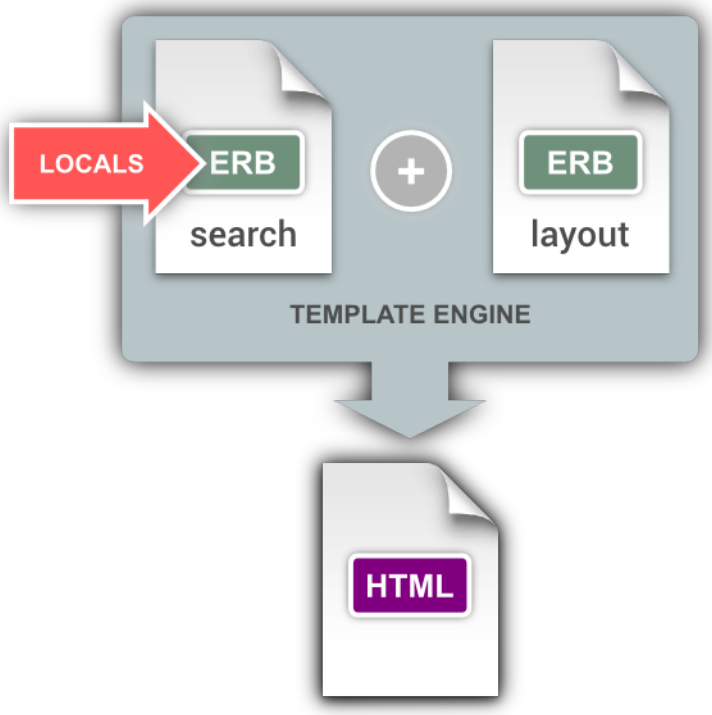

Figura 6.3: Proceso de construcción de una vista HTML

Las plantillas son esencialmente documentos HTML que incluyen código *Ruby* incrustado (`<%...%>`) con capacidad para acceder el contenido de las variables locales (*locals*) e invocar algunos métodos auxiliares (*helpers*) de la aplicación como se muestran en los siguientes ejemplos:

`layout.erb`

Determina la estructura básica de todas las vistas de la aplicación. En la sección `<head>` se define, entre otros, el título del documento, las hojas de estilo o la tipografía. En la sección `<body>` se define el elemento contenedor principal (`<div>`) dentro del cual se encuentra la sentencia `<%= yield %>`. Cuando esta es evaluada por el motor de plantillas, es sustituida por el contenido de otra plantilla, habitualmente un *parcial* como por ejemplo `:index`.

```
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title></title>
    <meta http-equiv="content-type" content="text/html;charset=utf-8"/>
    <link href="/css/reset.css" rel="stylesheet" type="text/css"/>
    <link href="/css/960.css" rel="stylesheet" type="text/css"/>
    <link href="/css/style.css" rel="stylesheet" type="text/css"/>
    <link href='http://fonts.googleapis.com/css?family=Dosis'
          rel='stylesheet' type='text/css'>
  </head>
  <body>
    <div class="container_12">
      <%= yield %>
    </div>
  </body>
</html>
```

`index.erb`

Es una plantilla de tipo *partial* que presenta el índice de fotografías. Por un lado muestra una serie de imágenes que enlazan con una serie de acciones como subir una nueva foto, realizar una búsqueda o mostrar las estadísticas. Por otro lado, itera sobre a la variable local `photos` (vector con información de cada fotografía) y construye el *collage* de fotografías.

```
<div style="text-align:right" class="grid_1 push_5">
  <a href="/photos/new"><img style="padding-top:10px"
     class="button" src="assets/upload.png"/></a>
  <a href="/search/photos"><img class="button" src="assets/search.png"/>
  </a>
```

```
  <a href="/stats/photos"><img class="button" src="assets/info.png"/>
  </a>
</div>

<div class="grid_5 push_5">
  <%photos.each do |i|%>
    <div style="float:left">
       <a href="<%=i[1]%>"/>
          <img class="thumbnail" src="<%=i[0]%>"/>
       </a>
    </div>
  <%end%>
</div>
```

Capítulo 7

Despliegue y control

7.1. Introducción

En los dos capítulos anteriores se han expuesto los detalles referentes al diseño e implementación de la aplicación. En éste, las secciones que se presentan a continuación describen, de forma ordenada, los pasos básicos necesarios para desplegar y controlar dicha aplicación sobre la plataforma *Heroku*.

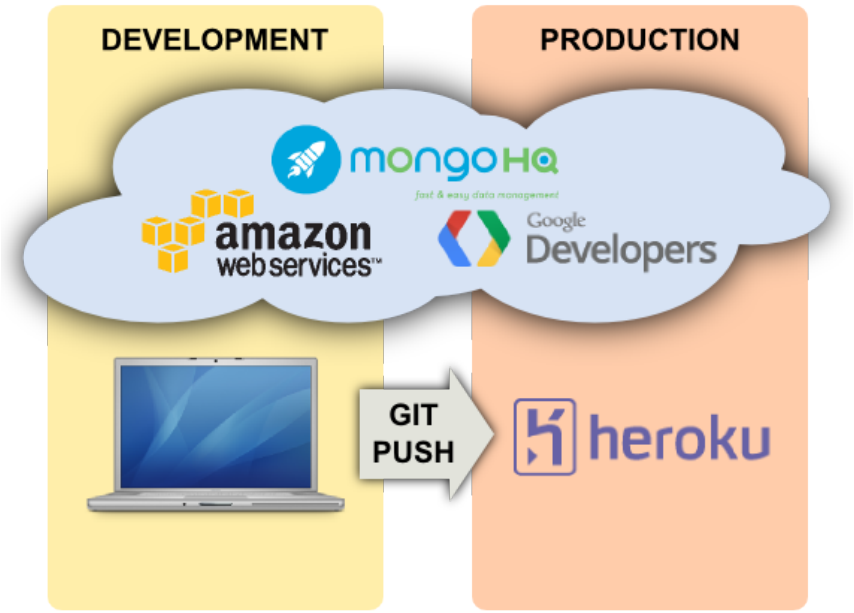

Figura 7.1: Transición del entorno de desarrollo al de producción (*Heroku*).

7.2. Configuración del equipo de desarrollo

Antes de implementar y desplegar aplicaciones en la plataforma *Heroku* es necesario instalar en los equipos los componentes *software* que se indican a continuación.

7.2.1. Entorno de programación

Heroku, como ya indicamos en el capítulo tercero, es una plataforma 'políglota' y soporta aplicaciones implementadas en cualquier lenguaje de programación, siempre y cuando exista un *buildpack* adecuado.

En el presente proyecto, la aplicación ha sido implementada sobre el entorno de programación *Ruby* y por ello será necesario instalar los siguientes elementos:

- **Ruby**: entorno de ejecución *Ruby* (versión 1.9) y sus bibliotecas básicas.
- **Rubygems**: gestor de paquetes o gemas (*gems*) del entorno de programación *Ruby*.
- **Bundler**: herramienta escrita en *Ruby* que simplifica el manejo de dependencias en las aplicaciones.

En sistemas operativos que disponen de gestor de paquetes la instalación de *Ruby* y *Rubygems* suele ser un proceso sencillo que solo requiere una instrucción en la consola de comandos como se muestra a continuación:

```
$ sudo port install ruby19        # Mac OS X + Macports
$ brew install ruby               # Mac OS X + Homebrew
$ sudo apt-get install ruby1.9.1  # Debian GNU/Linux y Ubuntu
$ pkg install runtime/ruby-18     # Solaris y OpenIndiana
```

Una vez que el entorno de programación *Ruby* está disponible, el siguiente paso es instalar la gema *Bundler* utilizando el comando `gem1.9 install GEMNAME`.

```
$ sudo gem1.9 install bundler
```

7.2.2. *Heroku Toolbelt*

Para poder crear aplicaciones y desplegar código sobre la plataforma *Heroku*, es necesario instalar el denominado *Heroku Toolbelt*[89] que engloba las siguientes herramientas:

- ***Heroku CLI***: herramienta que permite interactuar con *Heroku* para crear y controlar aplicaciones.
- **Foreman**: gestor de procesos para aplicaciones con múltiples componentes.

- **Git**: herramienta para gestión y control de versiones de código.

Heroku ha simplificado el proceso de instalación de estos tres elementos proporcionando paquetes específicos para diversos sistemas operativos, como *OS X*, *Windows* y varias distribuciones *Linux*.

Opcionalmente, estos elementos se pueden instalar por separado. Por un lado, *Heroku CLI* y *Foreman* utilizando, nuevamente, el comando `gem1.9 install`.

```
$ sudo gem1.9 install heroku foreman
```

Por otro, la instalación de la herramienta *Git*, como en el caso de *Ruby* y *Rubygems* mediante uno de los siguientes comandos:

```
$ sudo port install git-core     # Mac OS X + Macports
$ brew install git               # Mac OS X + Homebrew
$ sudo apt-get install git-core  # Debian GNU/Linux y Ubuntu
```

7.3. Cuenta de usuario y gestión de claves *SSH*

Otro requisito para operar en la plataforma *Heroku* es poseer una cuenta de usuario. Crear una es un proceso muy rápido y no tiene coste[90]. El siguiente paso será iniciar sesión mediante el comando `heroku login` indicando la dirección de *email* y la contraseña utilizadas durante la creación de la cuenta.

```
$ heroku login
Enter your Heroku credentials.
Email: username@gmail.com
Password (typing will be hidden):
Authentication successful.
```

A partir de este momento es posible realizar acciones en *Heroku*, como por ejemplo crear y destruir *apps*. Una aplicación (*app*) es la unidad fundamental de organización en *Heroku* y cada usuario puede crear un número ilimitado de ellas. Mediante el comando `heroku help` se obtiene una breve descripción de todas las acciones disponibles.

```
$ heroku help
Usage: heroku COMMAND [--app APP] [command-specific-options]

Primary help topics, type "heroku help TOPIC" for more details:
```

```
  addons    #  manage addon resources
  apps      #  manage apps (create, destroy)
  auth      #  authentication (login, logout)
  config    #  manage app config vars
  domains   #  manage custom domains
  logs      #  display logs for an app
  ps        #  manage processes (dynos, workers)
  releases  #  manage app releases
  run       #  run one-off commands (console, rake)
  sharing   #  manage collaborators on an app

Additional topics:

  account      #  manage heroku account options
  certs        #  manage ssl endpoints for an app
  db           #  manage the database for an app
  drains       #  display syslog drains for an app
  git          #  manage git for apps
  help         #  list commands and display help
  keys         #  manage authentication keys
  labs         #  manage optional features
  maintenance  #  manage maintenance mode for an app
  pg           #  manage heroku-postgresql databases
  pgbackups    #  manage backups of heroku postgresql databases
  plugins      #  manage plugins to the heroku gem
  ssl          #  manage ssl certificates for an app
  stack        #  manage the stack for an app
  status       #  check status of heroku platform
  update       #  update the heroku client
  version      #  display version
```

Heroku utiliza un mecanismo de criptografía asimétrica o de dos claves (pública y privada) para identificar qué usuarios están autorizados a desplegar código. El comando `ssh-keygen` permite generar dichas claves.

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/gonzalo/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/gonzalo/.ssh/id_rsa.
Your public key has been saved in /Users/gonzalo/.ssh/id_rsa.pub.
```

```
The key fingerprint is:
d3:b6:64:79:e3:90:33:f9:d5:49:d6:e3:04:d5:6c:3e gonzalo@Denver.local
```

La primera vez que se utiliza el comando `heroku`, la clave pública es actualizada en *Heroku*. Esto permite al usuario desplegar código en cualquiera de sus aplicaciones. Para obtener una lista de todas las claves públicas autorizadas se utiliza el comando `heroku keys`.

```
$ heroku keys
=== gonzalo.llorente@gmail.com Keys
ssh-rsa AAAAB3NzaC...LthjJRQWXj gonzalo@Denver.local
```

Puede que sea preciso añadir nuevas claves públicas, por ejemplo, para permitir a otros usuarios desplegar código. Esto se realiza mediante el comando `heroku keys:add [FILE]`.

```
$ heroku keys:add  # argumento por defecto "~/.ssh/id_rsa.pub"
Found existing public key: /Users/gonzalo/.ssh/id_rsa.pub
Uploading SSH public key /Users/gonzalo/.ssh/id_rsa.pub... done
```

Para eliminar una clave pública cuya seguridad se ha visto comprometida o que simplemente ya no es necesaria se utiliza el comando `heroku keys:remove [NAME]`.

```
$ heroku keys:remove gonzalo@Denver.local
Removing gonzalo@Denver.local SSH key... done
```

7.4. Declaración de dependencias (`Gemfile`)

La aplicación desarrollada en el presente proyecto tiene dependencias sobre un importante número de gemas. Controlar manualmente qué gemas y qué versión de cada una de ellas se encuentra disponible en los diferentes entornos de ejecución, a menudo, se convierte en una tarea tediosa. *Heroku*, como se indica en la sección 3.2.2, recomienda hacer uso de la herramienta *Bundler* para resolver e instalar dependencias tanto en los entornos de desarrollo y pruebas como en los de despliegue.

Para que *Bundler* pueda realizar su cometido, es necesario crear un fichero llamado `Gemfile` en la raíz del directorio del proyecto que describa qué gemas (y opcionalmente qué versión de éstas) requiere la aplicación. En el caso que nos ocupa, el fichero tendría el siguiente contenido:

```
source :rubygems    # valid Rubygems repository
gem 'sinatra'
gem 'bson_ext'
gem 'mongo'
gem 'mongo_mapper'  # gem 'mongo_mapper', '0.9.1' (version optional)
gem 'aws-s3'
gem 'exifr'
gem 'rmagick'
gem 'thin'
```

Mediante el comando `bundle check` se comprueba si todas las gemas necesarias están instaladas o falta alguna.

```
$ bundle check
The following gems are missing
 * sinatra (1.3.3)
Install missing gems with `bundle install`
```

En el ejemplo anterior, la gema *Sinatra* no está instalada. Para resolver este problema simplemente habría que ejecutar el comando `bundle install`.

```
Fetching gem metadata from http://rubygems.org/........
Using i18n (0.6.1)
Using multi_json (1.3.6)
[...]
Using tilt (1.3.3)
Installing sinatra (1.3.3)
Using thin (1.5.0)
Using bundler (1.2.1)
Your bundle is complete! Use `bundle show [gemname]` to see where a
bundled gem is installed.
```

La ejecución del comando `bundle install` genera un fichero llamado `Gemfile.lock`. Este fichero es utilizado por *Heroku* para asegurar que la versión de las gemas instaladas durante el despliegue de la aplicación son las mismas que las instaladas localmente en las máquinas de desarrollo y pruebas.

Una vez que las dependencias han sido resueltas e instaladas es posible ejecutar procesos *Ruby* que hagan uso de ellas. Para ello se utiliza el comando `bundle exec` que asegura que las gemas accesibles por los procesos y sus versiones son las especificadas en el fichero `Gemfile`. En la siguiente sección se muestra como se utiliza en la aplicación del proyecto.

7.5. Declaración de tipos de proceso (Procfile)

La declaración de los tipos de proceso no es un paso estrictamente necesario para desplegar aplicaciones escritas en la mayoría de los lenguajes soportados por *Heroku*. La plataforma automáticamente detecta el lenguaje y arranca un proceso por defecto de tipo `web` que ejecuta la aplicación desplegada. Sin embargo, la recomendación es declararlos explícitamente con el objetivo de tener un mayor control y flexibilidad sobre la aplicación.

Los tipos de proceso de una aplicación se declaran mediante un fichero llamado `Procfile` ubicado, al igual que el fichero `Gemfile`, en la raíz del proyecto. Cada línea define un tipo de proceso siguiendo el formato:

```
<process type>: <command>
```

- `<process type>`
  Cadena alfanumérica que identifica un tipo proceso. Por ejemplo: `web`, `worker`, `clock`, etc.
- `<command>`
  Comando asociado con el tipo de proceso. Por ejemplo: `ruby app.rb`.

Cada tipo de proceso declara el comando que debe ejecutarse cuando un proceso de ese tipo es invocado. Los tipos de proceso pueden llamarse de forma arbitraria excepto el tipo de proceso `web` que es especial porque es el único que recibe tráfico HTTP proveniente de la malla de encaminamiento de *Heroku*.

El contenido del fichero `Profile` de la aplicación desarrollada en este proyecto es el siguiente:

```
web: bundle exec ruby app.rb -p $PORT
```

En él se declara un proceso de tipo `web` que iniciará la ejecución de la aplicación mediante el comando `bundle exec ruby app.rb -p $PORT` tan pronto como se empiecen a recibir peticiones *web*.

Es conveniente comprobar que los entornos de ejecución locales que se utilizan durante las fases de desarrollo y pruebas sean iguales a los entornos remotos o de despliegue. Esto facilita detectar incompatibilidades y otro tipo de errores antes de desplegar una aplicación. Mediante el comando `foreman`, podemos iniciar la aplicación en un entorno local tal y como sucedería en uno remoto como *Heroku*.

```
$ foreman start
12:17:50 web.1  | started with pid 76395
12:17:53 web.1  | == Sinatra/1.3.3 has taken the stage on 5000 for
development with backup from Thin
```

Es posible que el nombre del intérprete *Ruby* en el sistema local no sea `ruby` sino `ruby1.9`, `ruby1.9.2` u otro distinto. O puede que sea necesario iniciar la aplicación en un puerto específico. En estos casos es posible crear un nuevo fichero llamado, por ejemplo, `Profile.dev` que tenga en cuenta estas diferencias.

```
web: bundle exec ruby1.9 app.rb -p 3000
```

La ejecución del comando `foreman` utilizando un fichero `Procfile` específico se realiza de la siguiente manera:

```
$ foreman start -f Procfile.dev
12:17:50 web.1  | started with pid 81302
12:17:53 web.1  | == Sinatra/1.3.3 has taken the stage on 3000 for
development with backup from Thin
```

7.6. Control de cambios (*Git*)

Git es un potente sistema de control de versiones distribuido y es el medio fundamental para desplegar aplicaciones en *Heroku*. Conocer en detalle su funcionalidad para interactuar con la plataforma no es estrictamente necesario, pero hacerlo reporta notables beneficios. A continuación se indican los comandos básicos necesarios.

Para comenzar a controlar los cambios en una aplicación mediante *Git* es preciso crear un repositorio local ejecutando el comando `git init` en la raíz del proyecto. Si el proyecto estuviera ubicado en el directorio *~/myapp* sería:

```
$ cd ~/myapp
$ git init
Initialized empty Git repository in /Users/gonzalo/myapp/.git/
```

El comando `tree` muestra en forma de árbol todos los archivos y directorios que componen la aplicación desarrollada en este proyecto.

```
$ tree
.
|- Gemfile
|- Gemfile.lock
|- Procfile
|- Procfile.dev
|- app.rb
|- public
```

```
|     |- css
|     |- 960.css
|     |- reset.css
|     |- style.css
|- views
      |- edit.erb
      |- layout.erb
      |- main.erb
      |- new.erb
      |- photo.erb
```

Mediante el comando `git add` es posible indicar qué archivos y directorios de los listados en el cuadro anterior es preciso controlar.

```
$ git add Gemfile Gemfile.lock Procfile app.rb public views
```

Por último, el comando `git commit -m [MESSAGE]` guarda una instantánea del estado de cada uno de los ficheros controlados en ese momento junto con el mensaje indicado.

```
$ git commit -m "First commit"
[master (root-commit) cdf612e] First commit
 19 files changed, 1020 insertions(+)
 create mode 100644 Gemfile
 create mode 100644 Gemfile.lock
 create mode 100644 Procfile
 create mode 100755 app.rb
 create mode 100755 public/css/960.css
 create mode 100755 public/css/reset.css
[...]
```

Obtener una lista de los cambios registrados por *Git* es posible con el comando `git log`.

```
$ git log --graph --pretty=oneline --abbrev-commit
* 36c45e0 Procfile tweak.
* 9f268cb Gemfile.lock added.
* a7a0204 Some functionality implemented.
```

7.7. Creación y despliegue de la aplicación.

Antes de poder desplegar una aplicación sobre la plataforma *Heroku* es necesario ejecutar el comando `heroku apps:create [NAME]`.

```
$ heroku apps:create
Creating powerful-brook-4212... done, stack is cedar
http://powerful-brook-4212.herokuapp.com/ | git@heroku.com:powerful-brook
-4212.git
Git remote heroku added
```

En este caso, no hemos proporcionado un nombre para aplicación y *Heroku* genera uno automáticamente. Esta es la opción más recomendable ya que el espacio de nombres es común para todas las aplicaciones y nombres como *news* o *blog* probablemente no estén disponibles.

La salida del comando anterior indica que la aplicación, una vez iniciada, se encontrará disponible en `http://http://powerful-brook-4212.herokuapp.com`. La segunda URL, `git@heroku.com:powerful-brook-4212.git` es la dirección del repositorio remoto *Git* en *Heroku*. Por defecto, el comando `heroku apps:create` añade automáticamente una entrada en el repositorio *Git* local de la aplicación denominada `heroku` que apunta a esta URL.

Finalmente, para desplegar la aplicación simplemente es necesario trasladar una copia del código fuente al repositorio *Git* de la aplicación en *Heroku* mediante el siguiente comando:

```
git push heroku master
```

7.8. Variables de configuración

Cada uno de los entornos de despliegue en *Heroku* dispone de su propio espacio de variables de entorno. Esta es una característica muy importante de la plataforma ya que permite que una única versión del código fuente pueda ser ejecutada en diferentes entornos con diferentes configuraciones.

Por ejemplo, sería posible ejecutar el código fuente de una aplicación sobre dos entornos, uno configurado para realizar la persistencia de datos sobre el servicio *MongoHQ* y otro configurado para hacerla sobre *MongoLab*.

Para obtener las variables de entorno se utiliza el comando `heroku config`:

```
$ heroku config
=== powerful-brook-4212 Config Vars
GEM_PATH: vendor/bundle/ruby/1.9.1
```

```
LANG:     en_US.UTF-8
PATH:     bin:vendor/bundle/ruby/1.9.1/bin:/usr/local/bin:/usr/bin:/bin
```

El entorno sobre el que se ha desplegado la aplicación desarrollada requiere la inclusión de la variable `RACK_ENV` con el valor `production`. Esta variable se consulta para realizar la inclusión del *middleware New Relic* (véase sección 6.3.1) y para determinar el uso del protocolo HTTPS sobre ciertos recursos (véase sección 6.4.2).

La gestión de variables de configuración se realiza mediante el comando `heroku config`:

```
$ heroku config:set RACK_ENV=production
Setting config vars and restarting powerful-brook-4212... done, v5
RACK_ENV: production
```

Cuando se alteran las variables de configuración de un entorno, la plataforma *Heroku* automáticamente reincida la aplicación.

7.9. Visualización del registro de eventos

7.9.1. Tipos de *logs*

Heroku recolecta tres categorías de *logs* para las aplicaciones:

- *App logs*: Flujos de salida de la aplicación. Esta categoría incluye eventos generados desde la aplicación, el servidor de la aplicación o sus bibliotecas. Para filtrar esta categoría se debe utilizar el parámetro `--source app`.
- *System logs*: Mensajes sobre acciones realizadas por la plataforma *Heroku* en favor de la aplicación como: reinicio de procesos fallidos, activar y desactivar un *web dyno* o servir una página de error cuando se produce un error en la aplicación. El filtro para esta categoría es `--source heroku`.
- *API logs*: Mensajes sobre tareas de administración realizadas por los desarrolladores que trabajan en la aplicación como: despliegue de nuevo código, cambio del número de procesos en ejecución (*scaling*) o la activación del modo de mantenimiento. El filtro para esta categoría es `--source heroku --ps api`.

7.9.2. Obtención de *logs*

Mediante el comando `heroku logs` es posible acceder a los últimos eventos. Por defecto se obtienen las 100 últimas líneas del registro, pero es posible obtener hasta las últimas

1500 líneas utilizando el comando `heroku logs -n <num_lines>`.

Adicionalmente, es posible ver el *log* de una aplicación en tiempo real de forma similar a como funciona el comando `tail -f`. Esto permite visualizar los eventos justo en el momento en el que ocurren facilitando tareas de detección y corrección de errores. El comando para visualizar *logs* de esta manera es `heroku logs -t`.

7.9.3. Formato

Cada línea se imprime de la siguiente manera:

```
timestamp source[process]: message
```

El significado de cada elemento se explica a continuación:

- `timestamp`: fecha y hora en la que la línea de *log* fue generada por el proceso o componente.
- `source`: todos los procesos de la aplicación (procesos web, procesos de segundo plano (*workers*) y procesos programados (*cron*) son identificados como `app`. Todos los componentes del sistema *Heroku* como el encaminador HTTP o el gestor de procesos son identificados como `heroku`.
- `process`: nombre del proceso o componente que generó la línea de *log*. Por ejemplo, `worker.3`. El encaminador HTTP de *Heroku* aparece identificado como `router`.
- `message`: contenido de la línea de *log*.

Parte IV

Evaluación, conclusiones y líneas futuras de trabajo

Capítulo 8

Evaluación

8.1. Introducción

Una vez que la aplicación ha sido desplegada sobre la plataforma *Heroku*, es conveniente asegurar que ésta es capaz de manejar de forma adecuada la carga generada por los usuarios. Existen varias herramientas de *benchmarking* (*Httperf*[91], *Ab*[92] y *Curl-loader*[93] entre otras) que permiten probar el rendimiento de una aplicación o servicio web simulando decenas de conexiones simultáneas. En este proyecto se utiliza una herramienta no tan conocida como las mencionadas anteriormente pero igualmente efectiva denominada *Siege*[94]. De forma complementaria, se emplea una solución *SaaS* para el control del rendimiento de aplicaciones web denominada *New Relic*[95].

En las siguientes secciones se detallan ambas herramientas, así como las pruebas realizas y los resultados obtenidos.

8.2. Herramientas

8.2.1. Siege

Descripción

Es una utilidad que permite realizar pruebas de rendimiento sobre aplicaciones y servicios web. Con ella es posible configurar un número determinado de usuarios simulados para que realicen peticiones de forma repetitiva contra un determinado servidor web. Tras cada ejecución, la herramienta proporciona un completo informe con datos muy valiosos para determinar el rendimiento como el tiempo total de la prueba, la cantidad de datos transferidos, el ratio de transacciones, etc. Véase la sección 8.2.1 para obtener una descripción completa de todos los datos y su significado.

Siege tiene tres modos de funcionamiento: regresión (cuando se utiliza mediante la herramienta *bombardment*), simulación, y fuerza bruta. Las pruebas realizadas utilizan los

últimos dos modos. A continuación se describen las opciones más habituales para invocar la herramienta.

Opciones más habituales

- `-c NUM, --concurrent=NUM (CONCURRENT)`

  Permite establecer el número de usuarios concurrentes simulados. Este número está limitado por los recursos disponibles de la máquina donde se ejecuta `siege`.

- `-i, --internet (INTERNET)`

  El tráfico de los usuarios simulados es generado de forma aleatoria utilizando las URLs definidas en un fichero. Esta opción es válida solo cuando se utiliza un fichero de URLs.

- `-d NUM, --delay=NUM (DELAY)`

  Cada usuario simulado *descansa* por un intervalo de tiempo aleatorio de entre 0 y `NUM` segundos entre una petición y la siguiente.

- `-b, --benchmark (BENCHMARK)`

  Ejecuta la prueba en modo `NO DELAY` para pruebas de rendimiento. Por defecto, cada usuario simulado es invocado con al menos 1 segundo de demora. Esta opción elimina esta característica. Es opción no está recomendada cuando se realizan pruebas de carga.

- `-r NUM, --reps=NUM, --reps=once (REPETITIONS)`

  Permite ejecutar `siege` un número determinado de veces. Si se utiliza la opción `--reps=once`, `siege` se ejecuta utilizando el fichero de URLs una sola vez. Importante: la opción `-t/--time` toma precedencia sobre `-r/--reps`.

- `-t NUMm, --time=NUMm (TIME)`

  Permite ejecutar la prueba durante un período determinado de tiempo. El formato es `NUMm` donde `NUM` es una unidad de tiempo y el modificador `m` puede ser `s/S`, `m/M` o `h/H` para segundos, minutos y horas respectivamente. Para lanzar `siege` durante 1 hora es posible utilizar cualquiera de las siguientes combinaciones: `-t 3600s`, `-t 60m` o `-t 1h`.

- `-f FILE, --file=FILE (FILE)`

  Esta opción permite determinar la ruta del fichero de URLs. Si no se especifica, el fichero por defecto es `SIEGE_HOME/etc/urls.txt`.

Estadísticas de rendimiento

- Transacciones (`Transactions`)

  Número total de solicitudes realizadas al servidor web. En el ejemplo, cada uno de los 25 usuarios simulados (`-c 25`) realiza 10 peticiones (`-r 10`) para un total de 250 transacciones. Es posible que el número de transacciones exceda el número de solicitudes especificadas. Esto es debido a que *Siege* cuenta cada una de las solicitudes realizadas contra el servidor lo que significa que *redirections* y *authentication challenges* cuentan como dos, no como uno. En este sentido, la herramienta sigue las especificaciones de HTTP e imita perfectamente el comportamiento de un navegador.

- Disponibilidad (`Availability`)

  Porcentaje de conexiones correctamente establecidas con el servidor. Es el resultado de conexiones fallidas (incluyendo *timeouts*) dividido por el número total de intentos de conexión.

- Tiempo transcurrido (`Elapsed time`)

  Duración de la prueba. *Siege* contabiliza este tiempo desde el instante en el que es invocado hasta el momento en el que se completa la última de las solicitudes.

- Datos transferidos (`Data transferred`)

  Suma de los datos transferidos por cada uno de los usuarios simulados. Incluye la información de las cabeceras y el contenido. Debido a que esta cifra incluye la información de las cabeceras, el valor indicado por *Siege* será mayor que el proporcionado por el servidor. En modo internet (`-i`) que utiliza URLs tomadas de forma aleatoria de un fichero de configuración, este número variará de una prueba a otra.

- Tiempo de respuesta (`Response time`)

  Tiempo medio que se empleó en responder a las solicitudes de cada usuario simulado.

- *Transaction rate*

  Número medio de transacciones que el servidor ha sido capaz de atender por segundo. Esto es el número total de transacciones divido por el tiempo transcurrido.

- Rendimiento (`Throughput`)

  Número medio de *bytes* transferidos por segundo desde el servidor a todos los usuarios simulados.

- Concurrencia (`Concurrency`)

  Número medio de conexiones simultáneas. Este número crece cuando el rendimiento del servidor decrece.

- Transacciones exitosas (`Successful transactions`)

  Número de veces que el servidor ha respondido con un código de estado HTTP inferior a `400`.

- Transacciones fallidas (`Failed transactions`)

  Número de veces que el servidor ha respondido con un código de estado HTTP igual o superior a `400` mas la suma de todas las conexiones fallidas incluyendo *sockets timeouts*.

- Transacción más larga (`Longest transaction`)

  Cifra más alta de tiempo que tomó atender un sola transacción de entre todas las realizadas.

- Transacción más corta (*Shortest transaction*)

  Cifra más baja de tiempo que tomó atender un sola transacción de entre todas las realizadas.

8.2.2. New Relic

Descripción

New Relic es una solución para el control del rendimiento de aplicaciones y servicios web que se ofrece mediante el modelo de servicio *SaaS*. Este sistema es capaz de analizar el funcionamiento de aplicaciones ubicadas tanto en la nube como en instalaciones propias. Ofrece integración total con algunos proveedores *PaaS* entre los que se encuentra *Heroku*.

Características

- Control del rendimiento del interfaz de usuario

  Control en tiempo real de los tiempos de carga de cada página, uso de red y procesamiento del DOM y *renderizado* de la páginas. Obtención de los tiempos de carga por tipo de navegador y área geográfica.

- Visibilidad de la aplicación en su conjunto

  Tiempos de respuesta acumulados e índice *Apdex* de satisfacción del cliente. Obtención de datos sobre el rendimiento de la aplicación medido en peticiones por minuto. Generación del mapa de la topología de la aplicación y resumen de operaciones de la base de datos.

- Control de métricas en el servidor

  Métricas para analizar la utilización de procesador, memoria, red y procesos en ejecución.

- Satisfacción del cliente

  Mecanismos para definir acuerdos de nivel de servicio (SLA). Identificación de patrones de bajo rendimiento en el navegador.

- Diagnóstico y solución de cuellos de botella

  Resumen de transacciones web más solicitadas y su rendimiento. Indicación de consultas SQL más lentas. Informes de errores.

Integración en la aplicación desarrollada

Heroku ofrece integración con *New Relic* mediante un *add-on*. Para instalarlo simplemente es necesario ubicarse en la raíz de la aplicación y ejecutar el siguiente comando:

```
$ heroku addons:add newrelic:standard
```

New Relic y *Heroku* han establecido dos niveles de servicio[1]: *standard* y *professional*. El primero de ellos es gratuito y es el utilizado en las pruebas de este capítulo.

Tras añadir *New Relic* como *add-on* de la aplicación, es necesario una serie de pasos manuales adicionales:

- Por un lado, es necesario incorporar en la aplicación el agente *Ruby* de *New Relic* que se encarga de recabar y enviar los datos del comportamiento de la aplicación. Para ello se incorpora la siguiente dependencia en el fichero `Gemfile`:

  ```
  gem 'newrelic_rpm'
  ```

  Después, es necesario ejecutar el comando `bundle install` para resolver la dependencia y finalmente se deben registrar los cambios realizados a los ficheros `Gemfile` y `Gemfile.lock` en el sistema de control de versiones *Git*.

- Por otro hay que incorporar a la aplicación un fichero de configuración para *New Relic*. La forma más sencilla de obtenerlo es ejecutando el siguiente comando en la raíz de la aplicación:

  ```
  curl https://raw.github.com/gist/2253296/newrelic.yml >
  config/newrelic.yml
  ```

  Este fichero controla la forma en la que *New Relic* obtiene y analiza los datos de rendimiento de la aplicación. Para este proyecto, no se han realizado cambios sobre él.

[1] https://addons.heroku.com/newrelic

- Finalmente, es preciso incorporar el agente en el código de la aplicación y hacer que éste se ejecute solo en entornos identificados como producción. Para ello, es necesario añadir las siguientes lineas en el fichero de la aplicación `app.rb`:

```
configure :production do
  require 'newrelic_rpm'
end
```

  Además, es necesario identificar el entorno de despliegue como producción. Desde la raíz de la aplicación se debe ejecutar el siguiente comando:

```
$ heroku config:add RACK_ENV=production
```

8.3. Pruebas

8.3.1. Capacidad

El objetivo de esta prueba es determinar el comportamiento de la aplicación en condiciones de carga extremas. Los parámetros utilizados en la herramienta `siege` han sido los siguientes:

- `-b` Elimina la demora entre peticiones (adecuado para realizar pruebas de carga).
- `-c<num>` Determina el número de usuarios concurrentes. En esta prueba, este valor es incrementado de forma paulatina (10, 20, 40, 50 y 120). El número máximo de usuarios concurrentes con el que se han realizado pruebas es 120 pues el sistema operativo (OS X 10.8) tiene una limitación en el número de descriptores/ficheros que se pueden abrir.
- `-t5m` La duración de cada prueba ha sido fijada en 5 minutos.
- `"http://<application-domain>/stats/photos"` es el recurso web sobre el que se ha realizado la prueba.

En la tabla 8.1 se recogen los datos obtenidos.

Siege

De los resultados recogidos en la tabla 8.1 el dato que más resalta es el tiempo medio de respuesta (*response time*). Como se puede observar, según se aumentan el número de usuarios concurrentes, esta cifra empeora de forma drástica.

Otro dato que resulta llamativo es que en las pruebas con 40 o más usuarios concurrentes se han registrado transacciones fallidas, esto es errores 500 lanzados por el servidor.

	-c10	-c20	-c40	-c80	-c120
Transactions (hits)	3565	3718	3688	3700	3213
Availability (%)	100.00	100.00	99.92	99.97	99.88
Elapsed time (s)	299.18	299.64	299.43	300.37	299.37
Data transferred (MB)	6.57	6.85	6.78	6.80	5.91
Response time (s)	0.84	1.61	3.23	6.40	10.95
Transaction rate (trans/s)	11.92	12.41	12.32	12.32	10.73
Throughput (MB/s)	0.02	0.02	0.02	0.02	0.02
Concurrency	9.97	19.93	39.76	78.89	117.53
Successful transactions	3565	3718	3688	3700	3213
Failed transactions	0	0	3	1	4
Longest transaction	3.85	12.25	9.40	14.95	22.83
Shortest transaction	0.38	0.41	0.50	0.46	0.53

Tabla 8.1: Resultados de las pruebas realizadas sobre la aplicación con *Siege*.

New Relic

Los gráficos ofrecidos por la herramienta *New Relic* corroboran aproximadamente los obtenidos mediante la herramienta *Siege*. En la figura 8.1 podemos sacar las siguientes conclusiones:

- El gráfico *app server response time* muestra como a lo largo de la prueba y según se van incrementando el número de usuarios concurrentes, el tiempo medio de respuesta va creciendo de forma paulatina desde los 500 ms hasta alcanzar picos que superan los 2100 ms. En estas cifras está computado el tiempo de acceso y espera por servicios auxiliares como *MongoHQ* en este caso.

- Dado que el tiempo medio de respuesta a lo largo de la prueba empeora hasta superar los 10 segundos el índice *Apdex* cae hasta situarse por debajo del 0.5, lo que indica un bajo grado de satisfacción.

- El gráfico que muestra la capacidad (*throughtput*) muestra sin equívoco como el máximo número de peticiones por minuto durante la prueba se alcanza con aproximadamente 720 respuestas por minuto.

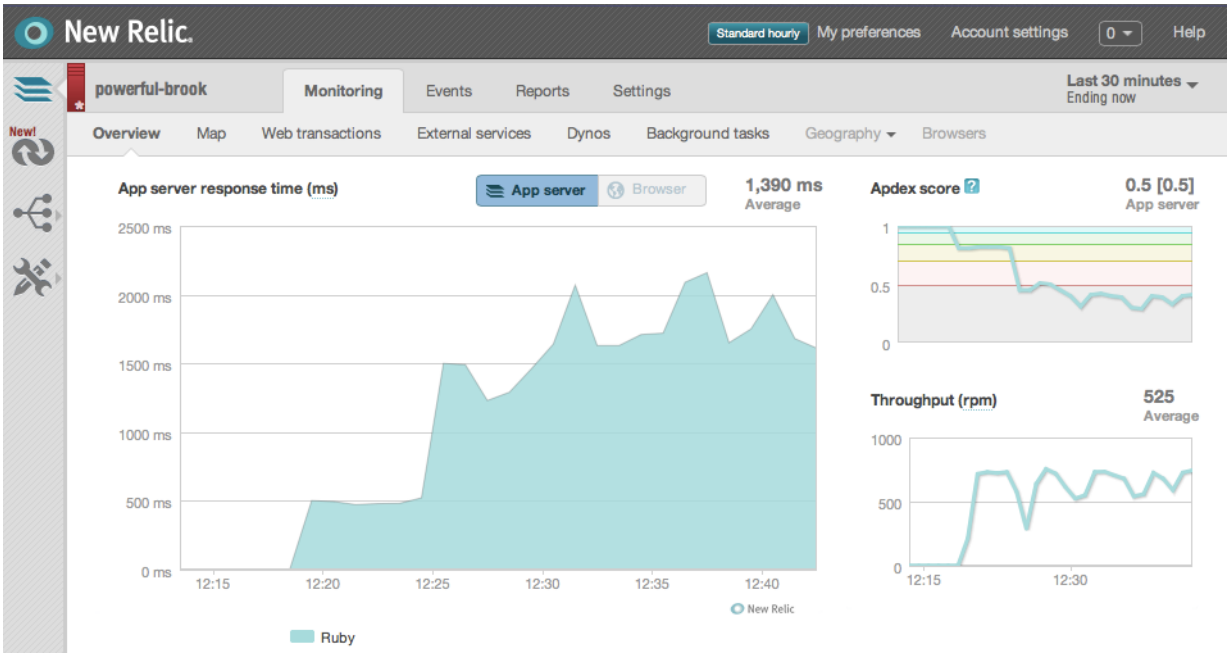

Figura 8.1: Pantalla *Monitoring Overview* de *New Relic*

Por otro lado, la figura 8.2 ofrece la siguiente información:

- La medida de memoria utilizada por el proceso web de la aplicación (*web dyno*) es de algo menos de 60 MB y es constante a lo largo de la prueba.

- En el gráfico *dynos running* se puede ver como durante la prueba se ha estado ejecutando solo un *dyno* y como la carga se ha ido incrementando sobre el hasta alcanzar un carga de 20 peticiones.

- En el gráfico *dyno restarts* se puede comprobar que durante la prueba no hubo ninguna parada del *dyno* en ejecución.

- Finalmente, *backlog* muestra que ninguna petición fue encolada por *Heroku* por falta de capacidad.

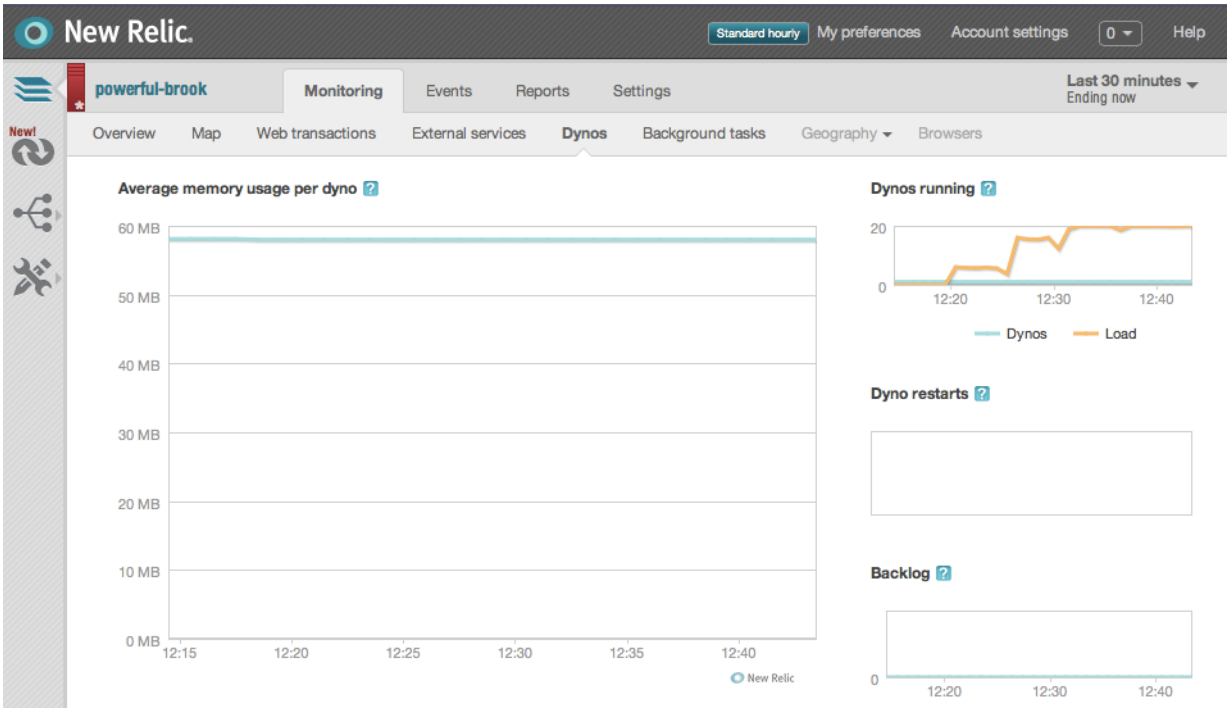

Figura 8.2: Pantalla *Monitoring Dynos* de *New Relic*

A la vista de los resultados, y teniendo en cuenta las limitaciones de la herramienta *Siege* para generar más de 120 usuarios concurrentes, podemos determinar que para el recurso analizado `http://<application-domain>/stats/photos` el tiempo de respuesta bien altamente determinado por el tiempo de espera para obtener los datos del servicio auxiliar *MongoHQ*.

8.3.2. Internet

En esta prueba el objetivo es determinar el comportamiento de la aplicación inmersa en unas condiciones de uso similares a las de un entorno de producción. Para simular estas condiciones la herramienta `siege` se ha configurado con los siguientes parámetros:

- `-c50`: el número de usuarios simulados concurrentes es 50.
- `-d10`: cada usuario simulado espera un máximo de 10 segundos entre una petición y la siguiente.
- `-t15m`: la prueba se realiza durante 15 minutos.
- `-i`: genera tráfico aleatorio basándose en un fichero de URLs.

- `-f siege_urls.txt`: determina el fichero de URLs que se utiliza en la prueba.

Desafortunadamente, *Siege* no dispone de la funcionalidad necesaria para simular la carga de archivos mediante formularios HTML (RFC 1867). Por este motivo, durante la prueba se ha realizado la carga y borrado de 30 fotografías utilizando un navegador web.

En esta ocasión, *Siege* y el navegador web serán simplemente meras herramientas para generar tráfico HTTP y los datos que van a ser analizados serán los que provengan de la herramienta de control de rendimiento *New Relic*.

Antes de realizar la prueba, los procesos de la aplicación fueron parados durante 15 minutos. Los datos mostrados por la herramienta *New Relic* hacen referencia a los últimos 30 minutos de ejecución.

Siege

A continuación se muestran los datos arrojados por la herramienta *Siege*. Estos resultados no son fidedignos ya que no reflejan las peticiones `POST` (carga de fotografías) y `DELETE` (borrado de fotografías) que han sido realizadas de forma independiente. Por este motivo, no se realiza ninguna valoración sobre ellos.

```
siege -c50 -d10 -t15m -i -f siege_urls.txt

Transactions:              7557 hits
Availability:              100.00 %
Elapsed time:              899.90 secs
Data transferred:          14.26 MB
Response time:             0.77 secs
Transaction rate:          8.40 trans/sec
Throughput:                0.02 MB/sec
Concurrency:               6.49
Successful transactions:   5580
Failed transactions:       0
Longest transaction:       16.12
Shortest transaction:      0.30
```

Monitoring Overview (New Relic)

La pantalla *Monitoring Overview* (figura 8.3) de *New Relic* ofrece los siguientes elementos informativos:

- *App server response time*. El tiempo de respuesta media en el instante en el que se finalizó el test fue de 40 ms. En el gráfico se puede observar como una parte

del tiempo corresponde a procesamiento *Ruby* (la propia aplicación) y otra parte es procesamiento web externo (posiblemente interacción con los servicios *Amazon S3* y/o *MongoHQ*).

- *Apdex score*. Este valor refleja el grado de cumpliendo del tiempo de respuesta fijado para las peticiones. El valor de tolerancia T elegido es 0.5 y el factor multiplicador es 4. El significado se recoge en la siguiente tabla:

Nivel	Multiplicador	Tiempo (T = 0.5)
Satisfecho	`<=T`	Igual o menos que 0.5 segundos
Tolerado	`>T, <=4T`	Entre 0.5 y 2 segundos
Frustrado	`>4T`	Más de 2 segundos

La interpretación del ratio *Apdex* es la siguiente:

  - 1.0 significa que todas la peticiones fueron satisfactorias.
  - 0.0 significa que todas las peticiones fueron frustradas.
  - Las peticiones toleradas satisfacen a medias. Por ejemplo, si todas las peticiones fueron tolerables, el ratio sería 0.5.

En la prueba, el ratio es 0.99 luego prácticamente todas las peticiones han sido atendidas en menos de 0.5 segundos y se cumple el nivel de satisfacción.

- *Throughput*. Muestra el número de peticiones HTTP atendidas por minuto. En la gráfica se puede observar que el número máximo de peticiones por minuto alcanzado fue algo menos de 600, y la media para los últimos 30 minutos está en 264.

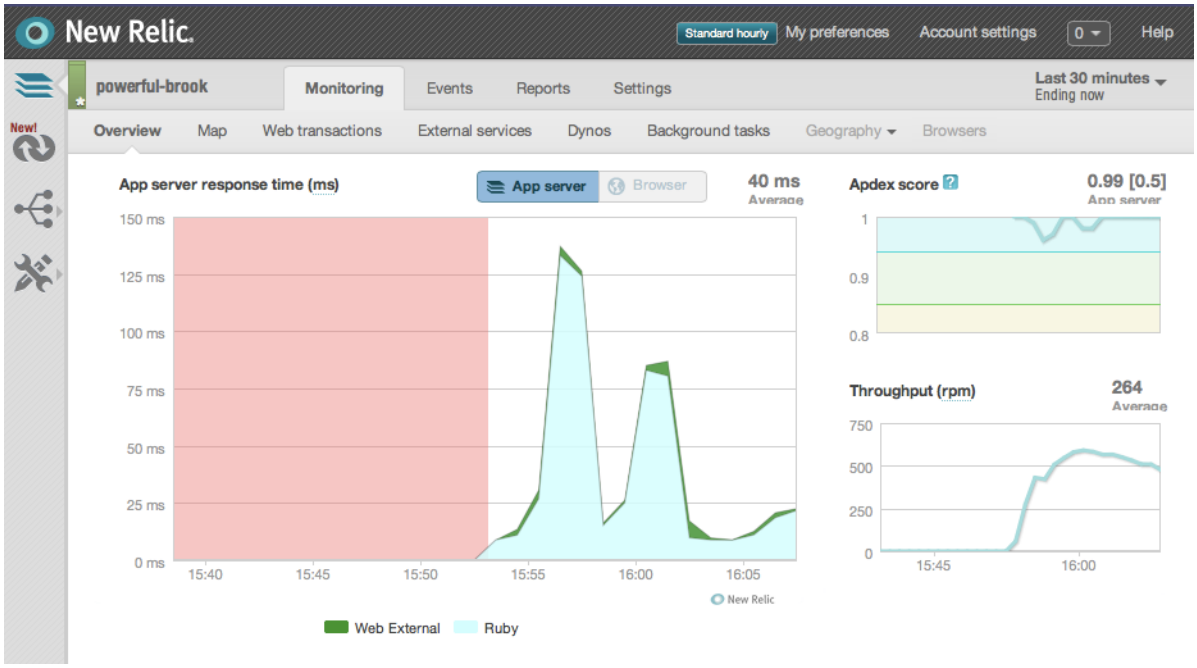

Figura 8.3: Pantalla *Monitoring Overview* de *New Relic*

Monitoring Map (New Relic)

La pantalla *Monitoring Map* (figura 8.4) de *New Relic* muestra la relación entre los diferentes elementos que forman la arquitectura de la aplicación. Cada elemento se muestra junto con su tiempo de respuesta, *throughput* y el estado. En el caso que nos ocupa, *New Relic* ha detectado dos elementos:

- powerful-brook (Aplicación)
  - Tiempo de respuesta: 40.9 ms
  - *Throughtput*: 264 cpm (*calls per minute*)
  - *Apdex*: 0.99
- funny-puffin.s3.amazonaws.com (Amazon S3)
  - Tiempo de respuesta: 108 ms
  - *Throughtput*: 6 cpm (*calls per minute*)
  - *Arr*: 1:42.6 (App request ratio)

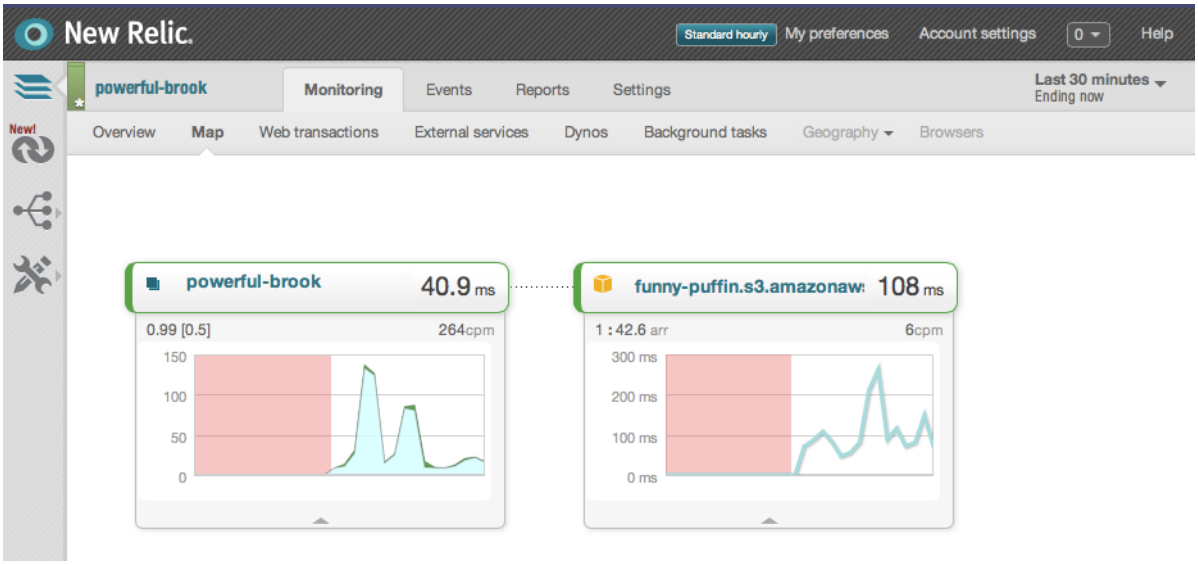

Figura 8.4: Pantalla *Monitoring Map* de *New Relic*

Monitoring Web Transactions (New Relic)

La pantalla *Monitoring Web Transactions* (figura 8.7) de *New Relic* ofrece los siguientes elementos informativos:

- Transacciones web ordenadas en base a diversos criterios.

  - En la figura 8.7 las transacciones web se muestran ordenadas de mayor a menor lentitud de respuesta. Como se puede observar, el recurso `POST /photos` que permite cargar nuevas fotografías en la aplicación es el más lento con una media de 418 ms por transacción. Las siguientes dos transacciones más lentas (alrededor de 110 ms) son `DELETE /photos/:id` y `GET /stats/photos`.

    En general es razonable que estas transacciones sean las más lentas ya que soportan más procesamiento que el resto, como el escalado de las imágenes, preparación de datos y una mayor interacción con los sistemas auxiliares.

  - La figura 8.5 muestra las transacciones ordenadas de mayor a menor *throughput*. En este caso, los recursos `GET /photos/:id` y `GET /search/photos` ofrecen un rendimiento similar de aproximadamente 90 respuestas por minuto.

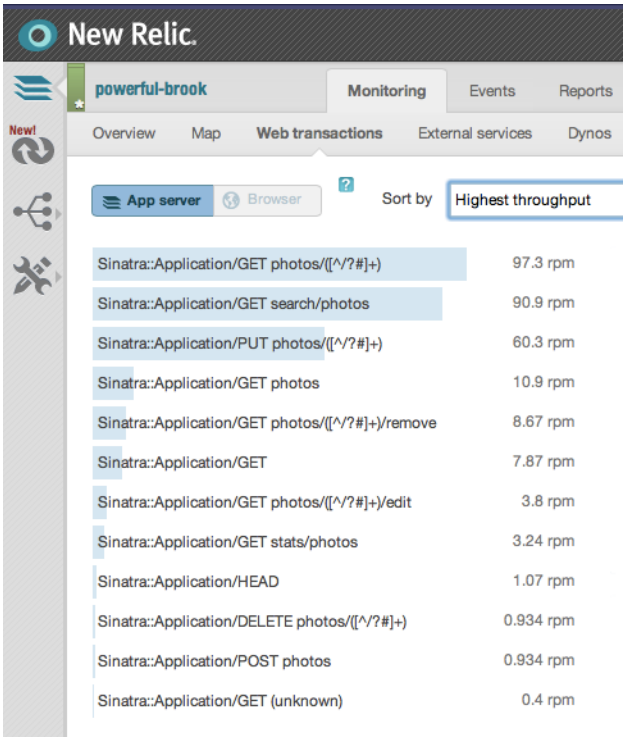

Figura 8.5: Transacciones web ordenadas por *throughput*.

- Por último, la figura 8.6 muestra primero las peticiones con peor índice *Apdex*. En este caso, los recursos `GET /photos` y `POST /photos` son los que peor índice *Apdex* registran.

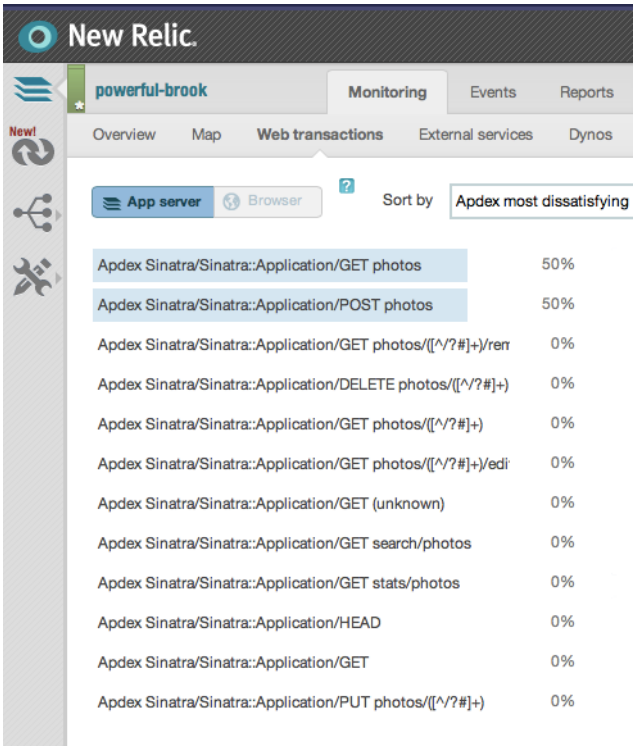

Figura 8.6: Transacciones web ordenadas por índice *Apdex*.

- *Top 5 web transactions by percentage of wall clock*. En este gráfico es posible observar qué transacciones han sido las que más recursos han consumido y su distribución a lo largo del tiempo. Por ejemplo, en la prueba realizada se puede determinar que alrededor de las 22:25 horas, la transacción que más recursos estuvo consumiendo fue `POST /photos`.

- *Response time and throughput*. En este gráfico es posible observar cronológicamente, el tiempo medio de respuesta de la aplicación (todos los recursos) junto con el *throughput* (capacidad). Como se puede ver en el gráfico, durante la prueba el tiempo medio de respuesta siempre fue inferior a 25 ms y la capacidad se situó alrededor de las 600 peticiones atendidas por minuto.

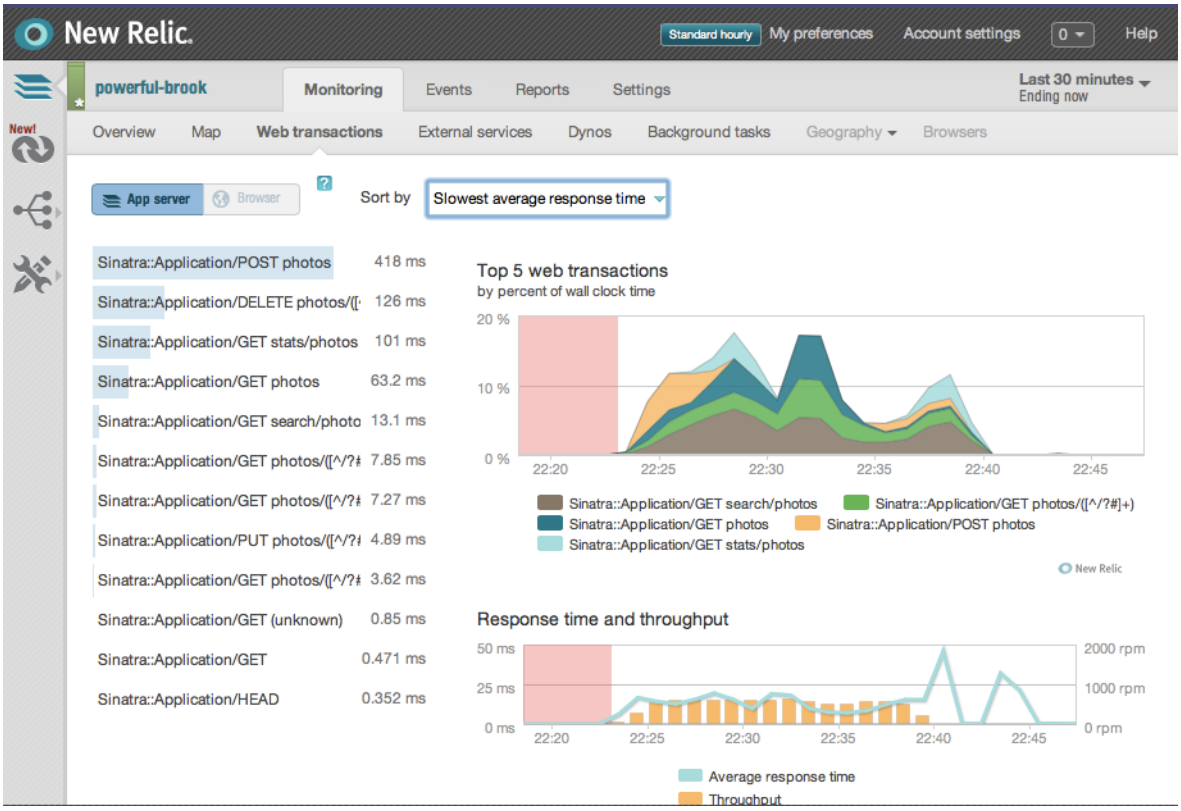

Figura 8.7: Pantalla *Monitoring Web Transactions* de *New Relic*

New Relic - Dynos

La pantalla *Monitor Dynos* (figura 8.8) de *New Relic* ofrece los siguientes elementos informativos:

- *Average memory access per dyno*: muestra la media de uso de memoria por cada *dyno*. En la prueba, el consumo de memoria se ha ido incrementando rápidamente hasta quedarse estable en 80 MB.

- *Dynos running*: muestra el número de *dynos* en uso junto con la carga. El número de *dynos* es siempre 1. También es posible apreciar como en dos momentos se han producido dos picos de carga. Posiblemente estos picos se correspondan con la carga masiva de fotografías.

- *Dynos restarts*: muestra el el número de paradas e inicios de los *dynos*. En la gráfica se puede observar que se produce un reinicio de un *dyno* cuando comienza la prueba.

- *Backlog*: cuando el número de peticiones concurrentes es mayor que el número de *dynos* que se están ejecutando, *Heroku* construye una cola de procesamiento deno-

minanda *backlog*. Este gráfico muestra su evolución. Durante la prueba el *backlog* siempre se ha mantenido vacío.

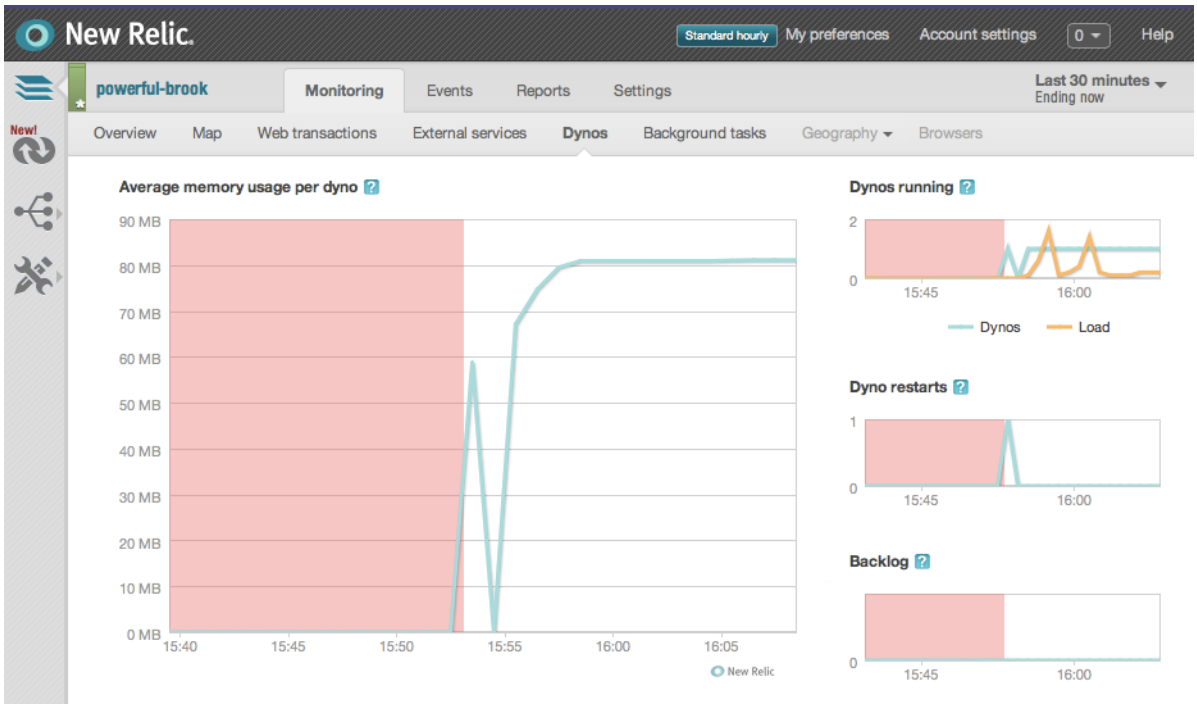

Figura 8.8: Pantalla *Monitoring Dynos* de *New Relic*

Capítulo 9

Conclusiones y líneas futuras de trabajo

En este último capítulo se presentan, por un lado, las conclusiones sobre los objetivos enunciados en el primer capítulo, evaluando su nivel de satisfacción y cumplimiento, y por otro, se enumeran algunos aspectos del proyecto que no han podido ser evaluados en detalle y cuyo estudio se recomienda para proyectos futuros.

Como adelanto a las siguientes secciones se puede afirmar que la impresión general sobre el proyecto realizado es positiva y se han cumplido en buena medida los objetivos marcados. El resultado final es un trabajo que abarca temas actuales concernientes con el diseño, implementación y puesta en producción de aplicaciones y servicios web utilizando las más novedosas tecnologías y herramientas *cloud computing*.

9.1. Conclusiones

9.1.1. Conclusiones sobre los objetivos

El presente proyecto, como se menciona en la sección 1.2, tiene como principal objetivo el diseño, implementación y despliegue de una aplicación para almacenar y mostrar fotografías utilizando tecnologías *cloud computing*. Para alcanzar dicho objetivo, se definieron otros más específicos los cuales se van a valorar de forma individual a continuación:

- El objetivo principal es la implementación y despliegue de una sencilla aplicación para almacenar y mostrar fotografías utilizando varios servicios en la nube. A lo largo de los capítulos quinto, sexto y séptimo se detallan todos los aspectos relacionados con la definición de la funcionalidad, diversos aspectos de la implementación e interacción con bibliotecas y servicios auxiliares y la configuración y despliegue final en la plataforma en la nube.

- El segundo objetivo ha sido presentar qué son las tecnologías *cloud computing*, cuáles son sus características esenciales y por qué son interesantes para los departamentos

de sistemas de información de las grandes corporaciones y para pequeñas empresas de desarrollo de aplicaciones. Este objetivo se cumple de forma exhaustiva a lo largo del segundo capítulo donde, entre otros, se ofrecen varios casos de éxito de empresas que han optado por este tipo de tecnologías. De forma adicional, en los capítulos tercero y cuarto se describen con detalle varios de los servicios en la nube que se han utilizado en el proyecto.

- Los objetivos tercero y cuarto se centran en la verificación de la funcionalidad y del rendimiento de la aplicación así como en el análisis de los resultados. En el capítulo octavo se presentan dos potentes herramientas con las cuales se han diseñado una serie de pruebas y que han permitido recoger y analizar datos sobre el comportamiento de la aplicación ante diversas situaciones de carga.

9.1.2. Conclusiones generales

Durante la realización del presente proyecto, el autor ha tenido ocasión de utilizar múltiples tecnologías como lenguajes de programación, bibliotecas, *frameworks* y servicios en la nube entro otros. A continuación se realizan algunas valoraciones sobre todos ellos:

- El *framework* de desarrollo utilizado *Sinatra* y el lenguaje de programación *Ruby* se combinan de forma excelente y permiten desarrollar aplicaciones y servicios web de una forma muy rápida y productiva. Al contrario que otros *frameworks*, *Sinatra* es agnóstico en cuanto a los mecanismos que deben proporcionar persistencia de datos o memoria inmediata por lo que el arquitecto de *software* tiene total libertad para elegir aquel que mejor se ajuste a las necesidades. Por otro lado, la simplicidad y elegancia de *Ruby* y su enorme cantidad de bibliotecas disponibles le convierten en una de las herramientas más productivas que he utilizado hasta la fecha.

- La plataforma *Heroku* ofrece un entorno muy bien diseñado para guiar todo el proceso de desarrollo y despliegue del código fuente. La integración que ofrece con *Git* es excelente y para desplegar la aplicación basta con un solo comando. La documentación que proporcionan está bien escrita y actualizada y su servicio de atención resuelve dudas y problemas con celeridad. Una de las ventajas que tiene frente a otros servicios de similares características es que ofrecen un nivel de servicio gratuito que permite que aplicaciones sencillas, como la realizada en este proyecto, puedan ejecutarse sin coste por tiempo indefinido.

- Por último, la herramienta *New Relic* sorprende y destaca por sus enormes capacidades para introducirse en el funcionamiento interno de la aplicación y ofrecer unos gráficos claros y precisos que permiten detectar funcionamientos anómalos. Es posible utilizarla en aplicaciones y servicios web implementados en casi cualquier lenguaje moderno e incluso dispone de agentes para controlar el funcionamiento de servidores.

9.2. Líneas futuras de trabajo

El presente proyecto proporciona una firme aproximación a las tecnologías y herramientas más relevantes que facilitan la implementación y despliegue de aplicaciones y servicios web en la nube. Sin embargo, el ámbito de *cloud computing* es muy amplio y evoluciona con enorme rapidez.

En las secciones siguientes se describen algunas de las áreas sobre las cuales sería interesante profundizar y que completarían el presente trabajo.

9.2.1. Múltiples tipos procesos

La aplicación que se ha desarrollado en este proyecto solo cuenta con un tipo de proceso, denominado *web dyno*, responsable de recibir la petición web, procesarla y generar la respuesta. Dado que la plataforma *Heroku* proporciona de forma gratuita 750 horas de *dynos* al mes por aplicación, este diseño permite que la aplicación pueda funcionar sin coste de forma permanente.

Sin embargo, no es el mejor diseño. Los procesos *web dyno* deben ser ligeros y su tiempo de procesamiento debe ser mínimo para que el cliente web espere el menor tiempo posible. Todos los trabajos pesados, como puede ser el cambio de tamaño de una imagen o la comunicación con servicios auxiliares debe hacerse siempre que sea posible como un trabajo en segundo plano utilizando otro tipo de procesos denominados *worker dynos*.

Sería interesante analizar qué funcionalidad de esta aplicación puede realizarse como trabajos en segundo plano y realizar su implementación.

9.2.2. Escalado horizontal

La plataforma *Heroku* ofrece mecanismos sencillos para dotar a las aplicaciones que se ejecutan sobre ella de recursos adicionales cuando la demanda lo exige. Tomando como punto de partida la sección anterior, una vez implementada la aplicación utilizado trabajos en segundo plano, sería interesante realizar pruebas de carga y evaluar su rendimiento utilizando diferentes configuraciones de tipos de procesos (n *web dynos* + m *worker dynos*).

9.2.3. Otros lenguajes y *frameworks*

El lenguaje de programación empleado en este proyecto es *Ruby* y el *framework* o biblioteca que facilita la creación de la aplicación web es *Sinatra*. Existen otras muchas opciones, algunas de las cuales se enumeran a continuación:

- *Ruby* y *Rails*
- *Node.js* y *Express*

- *Python* y *Django*

Sería interesante analizarlas y evaluar su idoneidad para realizar todo tipo de aplicaciones y servicios web.

9.2.4. Interfaz XML/JSON

La aplicación desarrollada ofrece solo un interfaz HTML apto para ser consumido por humanos utilizando navegadores web. Sin embargo, hoy en día, cada vez es más habitual que las aplicaciones y servicios web sean accedidos por otros sistemas como aplicaciones nativas en teléfonos móviles o incluso por otros servicios.

Para este tipo de situaciones, sería recomendable modificar la aplicación para que los recursos pudieran admitir y ofrecer la información en base a documentos XML o JSON.

9.2.5. Otras plataformas y servicios en la nube

Como ya se expuso en el capítulo 3, *Heroku* no es la única plataforma para desplegar aplicaciones y servicios web. De la misma forma, los servicios auxiliares utilizados por la aplicación implementada en este proyecto *Amazon S3* y *MongoHQ* son solo algunos de los muchos disponibles.

La decisión sobre cuál de ellos utilizar puede estar determinada por varios aspectos como la capacidad de procesamiento, su facilidad de uso o el coste entre otros. Sería interesante disponer de un estudio organizado por categorías de tipo de servicio (almacenamiento de datos, memoria inmediata, análisis de rendimiento, envío de correo electrónico y SMS o sistemas de colas de mensajes) que determinara en base a una serie de criterios cuál de los proveedores es el más adecuado para un proyecto en particular.

9.2.6. *New Relic*

New Relic se ha convertido en los últimos tiempos en una de las herramientas de referencia para el control de rendimiento de aplicaciones y servicios web. En este proyecto solo se han utilizado algunas de las opciones disponibles en la versión gratuita que ofrece el *add-on* de *Heroku*. Es por tanto interesante evaluar qué herramientas adicionales ofrece la versión de pago.

Parte V

Apéndices

Apéndice A

Presupuesto

En este apéndice se analiza el desarrollo de este proyecto en términos económicos.

A.1. Tareas

En esta primera sección se muestra una estimación de la duración de las distintas tareas llevadas a cabo durante el desarrollo de este proyecto. Éstas no han sido realizadas de forma continua ya que el desarrollo del proyecto se ha compaginado con actividad laboral.

La tabla A.1 recoge las tareas realizadas junto con el número de horas estimadas para su finalización.

Tarea	Horas
Estudio del lenguaje *Ruby* y la biblioteca *Sinatra*	60
Estudio de la plataforma *Heroku*	60
Estudio de los servicios *Amazon S3*, *MongoHQ* y *Google Image Charts*	80
Definición de la funcionalidad	20
Diseño de la solución	40
Elaboración del código fuente	180
Diseño del conjunto de pruebas	40
Realización de pruebas y análisis de resultados	80
Creación de la memoria del proyecto	60

Tabla A.1: Tareas del proyecto

A.2. Recursos

A.2.1. Infraestructura

Para la realización del proyecto se han utilizado principalmente dos infraestructuras con repercusión en los costes finales: el local de trabajo y la infraestructura de comunicaciones.

Respecto al local, en concepto de alquiler, limpieza y mantenimiento se han establecido unos costes de 6.000€/año, que al ser compartido durante el proyecto con tres personas suponen un total de 2.000€/año.

La infraestructura de comunicaciones consiste en la conexión de red a Internet. Se estima que para la elaboración del presente proyecto y dado que se hace un uso moderado del enlace (descargas desde Internet de *software* y documentación), una solución ADSL de unos 10Mbit/s es suficiente. Este acceso tiene un coste de unos 480€/año.

Ambos recursos consumidos tienen un coste total de **2480€/año**.

A.2.2. Recursos materiales

Durante el desarrollo del proyecto se ha utilizado solo una máquina. La utilización de los elementos de red que han sido necesarios (punto de acceso a la red inalámbrica y *router*) no se valoran por estar incluidos dentro del coste de la infraestructura de comunicaciones.

Ordenador portátil

- Modelo: *Apple MacBook Pro 3,1*
- Procesador: *2.2 GHz Intel Core 2 Duo*
- Video: *NVIDIA GeForce 8600M GT 128 MB*
- Memoria RAM: *4 GB 667 MHZ DDR2 SDRAM*
- Capacidad de disco duro: *120 GB*
- Interfaz de red: *Marvell Yukon Gigabit Adapter, AirPort Extreme*
- Duración: Todo el proyecto
- Coste de adquisición: 1680€
- Valor marginal: 500€
- Coste imputable: **1200€/año** (régimen de alquiler)

A.2.3. Recursos *software* y servicios

Durante el proyecto se han utilizado varios elementos software como sistemas operativos, lenguajes de programación, *frameworks*, editores de texto y gráficos así como numerosos servicios en la nube. A continuación se enumeran algunos de los más relevantes, aunque ninguno ha supuesto coste alguno.

Coste de adquisición de sistemas operativos

- *Mac OS X 10.8.2* (0€, ya que se proporciona sin coste con la adquisición del equipo)

Coste de adquisición de *software*

- Entorno de programación *Ruby* y bibliotecas (0€)
- Editor de texto *VIM* (0€)
- Editor de gráficos vectoriales *Inkscape* (0€)
- Sistema de composición tipográfica LaTeX(0€)
- Gestor de paquetes *MacPorts* (0€)
- Herramienta *siege* (0€)
- Herramienta para realizar presentaciones *Keynote '09 (Apple)* (15€)

Coste de uso de servicios y plataformas en la nube

- Plataforma de aplicaciones *Heroku* (0€)
- Servicio de almacenamiento *Amazon S3* (0€)
- Servicio de base de datos *MongoHQ* (0€)
- Servicio *Google Image Charts* (0€)
- Servicio para control de rendimiento *New Relic* (0€)

Los recursos de *software* y servicios en la nube consumidos suman un coste total de **0€**.

A.2.4. Recursos humanos

Para la realización del proyecto se requiere la participación de diversos especialistas los cuales se recogen en la tabla A.2 junto con su coste aproximado por hora de trabajo.

Cargo	Código	Coste (€/hora)
Arquitecto	A	50,00
Desarrollador	D	30,00
Calidad y pruebas	C	35,00

Tabla A.2: Retribución de cada especialista

El conjunto de los costes imputables al consumo de recursos humanos se resume en la tabla A.3

TAREA	Sem.	Factor	Importe (€)
Fase 1. Estudio de tecnologías			
Lenguaje *Ruby* y la biblioteca *Sinatra*	1,5	A:0,8 D:0,2	2.760,00
Plataforma *Heroku*	1,5	A:0,8 D:0,2	2.760,00
Servicios *Amazon S3*, *MongoHQ* y *Google Image Charts*	2	A:0,8 D:0,2	3.680,00
Fase 2. Diseño e implementación			
Definición de la funcionalidad	0,5	A:0,9 D:0,1	960,00
Diseño de la solución	1	A:0,7 D:0,3	1.760,00
Elaboración del código fuente	4,5	A:0,1 D:0,9	5.760,00
Fase 3. Despliegue, pruebas y evaluación			
Diseño del conjunto de pruebas	1	A:0,2 C:0,8	1.520,00
Realización de pruebas y análisis de resultados	2	A:0,2 C:0,8	3.040,00
Fase 4. Documentación			
Creación de la memoria del proyecto	1.5	A:0,5 D:0,25 C:0,25	2.475,00
Subtotal			24.715,00

Tabla A.3: Presupuesto recursos humanos

A.3. Resultado de la planificación

La duración del proyecto se estima en 15.5 semanas (casi 4 meses) en jornadas de 8 horas durante 5 días por semana.

A.4. Resumen de costes del proyecto

A.4.1. Coste total

A continuación se calcula la parte proporcional de los costes por el alquiler de la infraestructura y los recursos materiales (*hardware*) convirtiendo el coste €/año a €/semana según el factor (7/365).

Recurso material	Coste (€/año)	Semanas	Importe (€)
Infraestructura	2.480,00	15,5	737,20
Hardware	1.200,00	15,5	356,71
Subtotal			1093,91

Teniendo en cuenta todos los costes anteriormente relatados, el coste total para realizar el proyecto asciende a **25.808,91€**, tal y como se recoge en la tabla A.4.

Concepto	Importe (€)
Infraestructura	737,20
Hardware	356,71
Software	15,00
Recursos humanos	24.715,00
Total	**25.823,91**

Tabla A.4: Coste de realización del proyecto

A.4.2. Presupuesto

Finalmente, en la tabla A.5 se desglosa el balance final del coste del proyecto donde se tienen en cuenta los porcentajes de beneficio, riesgo e impuestos.

Concepto	Importe (€)
Coste total	25.823,91
Riesgo (20 %)	5.161,78
Beneficio (20 %)	5.161,78
Subtotal	36.147,47
IVA (21 %)	7.590,97
Total	**43.738,44**

Tabla A.5: Presupuesto del proyecto

El coste total del proyecto asciende a **CUARENTA Y TRES MIL SETECIENTOS TREINTA Y OCHO EUROS CON CUARENTA Y CUATRO CÉNTIMOS**.

Madrid, 20 de marzo de 2013

El ingeniero proyectista.

Fdo. Gonzalo Suárez Llorente

Apéndice B

Metodología *Twelve-Factor App*

B.1. Introducción

En los últimos tiempos, las aplicaciones se diseñan para ofrecerse como un servicio (*SaaS*). La metodología *twelve-factor app* permite la construcción de aplicaciones que:

- Automatizan los procesos de configuración mediante formatos declarativos lo que permite minimizar el tiempo y coste cuando se incorporan al proyecto nuevos desarrolladores.
- Tienen un contrato claro con el sistema operativo subyacente lo que ofrece una flexibilidad máxima a la hora de ejecutarse en otros entornos (*maximum portability*).
- Son adecuadas para desplegarse sobre plataformas en la nube modernas evitando la necesidad de lidiar con servidores y la administración de sistemas.
- Minimizan las diferencias entre los entornos de desarrollo y producción permitiendo despliegue continuo (*continuous deployment*) para una agilidad máxima.
- Y pueden adaptarse ante un incremento de la demanda (*scale up*) sin requerir cambios bruscos en las herramientas, la arquitectura o las prácticas habituales de desarrollo.

Esta metodología es válida para aplicaciones escritas en cualquier lenguaje de programación y que utilizan cualquier tipo de servicio adicional (base de datos, *middleware* de mensajería, caches, etc).

B.2. Factores

B.2.1. Base de código (*Codebase*)

El código fuente de una aplicación que sigue las directrices de la metodología *twelve-factor app* siempre se gestiona mediante un sistema de control de versiones como *Git*,

Mercurial o *Subversion*. Estos sistemas registran cada cambio en el *software* en un repositorio. Una base de código se corresponde con un solo repositorio.

Existe únicamente una relación uno a uno entre la base de código y la aplicación:

- Si existen varias bases de código no estamos ante una aplicación sino un sistema distribuido. Cada elemento dentro de un sistema distribuido es en si mismo una aplicación y puede cumplir de forma individual con las indicaciones de *twelve-factor app*.
- Varias aplicaciones que comparten la misma base de código es una violación de la metodología. La solución es extraer la funcionalidad compartida en bibliotecas e incluirlas en cada aplicación mediante un gestor de dependencias (*dependecy manager*).

Solo hay una base de código por aplicación, pero habrá múltiples despliegues de la misma. Un despliegue es una instancia en ejecución de una aplicación (entorno de producción y varios de pruebas o *staging*). Adicionalmente, cada programador tiene una instancia en su entorno local de desarrollo.

El código base es el mismo para todos los despliegues, aunque es posible que las versiones activas en cada despliegue sean distintas. Por ejemplo, un programador puede tener implementada cierta funcionalidad no desplegada todavía en el entorno de pruebas y pruebas puede incluir cierta funcionalidad aun no presente en el entorno de producción. Pero todos ellos comparten un mismo código base permitiendo identificar diferentes despliegues con la misma aplicación.

B.2.2. Dependencias (*Dependencies*)

La mayoría de los lenguajes de programación ofrecen herramientas para empaquetar y distribuir bibliotecas como *CPAN* para *Perl* o *Rubygems* para *Ruby*. Existen dos modos de instalación de bibliotecas mediante estos mecanismos:

- Por equipo o *system-wide*
- Por aplicación o *scoped*

Las aplicaciones *twelve-factor app* nunca dependen de la existencia de bibliotecas instaladas a nivel de sistema (*system-wide*) y declaran sus dependencias de forma explícita y unívoca mediante un manifiesto. Utilizan además mecanismos de aislamiento que evitan que las bibliotecas del sistema interfieran durante la ejecución de la aplicación. La especificación de dependencias es uniforme para todos los despliegues (desarrollo, test o *staging* y producción).

Por ejemplo, la herramienta *Bundler* (en el entorno de programación *Ruby*) ofrece, por un lado el fichero *Gemfile* para declarar las dependencias y por otro el comando `bundle`

`exec` proporciona aislamiento de dependencias en ejecución (*dependency isolation*).

Uno de los beneficios de la declaración explícita de dependencias es que simplifica la configuración inicial de la aplicación, algo esencial para nuevos programadores que se incorporan al proyecto. Una vez que el entorno de programación y el gestor de dependencias están instalados, el programador solo tiene obtener una copia de la base de código y ejecutar un comando para construir la aplicación. Por ejemplo, este comando para el entorno *Ruby/Bundler* es `bundle install`.

Las aplicaciones *twelve-factor* tampoco pueden asumir la existencia de herramientas del sistema, por ejemplo, *ImageMagick* o *curl*. Aunque muchas de ellas habitualmente están presentes no existe garantía de que existan en todos los sistemas donde se va a desplegar la aplicación por lo que la recomendación es incorporar dichas herramientas dentro de la aplicación (*vendoring*).

B.2.3. Configuración (*Config*)

La configuración de una aplicación es todo aquello que es probable que varíe entre unos despliegues y otros (desarrollo, *staging* o producción). Esto incluye, por ejemplo:

- Configuración de base de datos, *Memcached* y otros servicios de persistencia.
- Credenciales de acceso a servicios externos como Amazon S3 o Twitter.
- Valores que dependen del entorno de despliegue.

En ocasiones, las aplicaciones almacenan en código (mediante constantes) aspectos de la configuración. Esto es una violación de la metodología que requiere una estricta separación entre configuración y código. La configuración varía notablemente entre despliegues mientras que el código fuente no. En otras ocasiones, la configuración es almacenada en archivos especiales que no son registrados en el sistema de control de versiones. Esta opción es válida pero no está exenta de problemas. Por ejemplo, es relativamente sencillo almacenar por error uno de estos archivos de configuración en el repositorio de código.

En la metodología *twelve-factor app*, la configuración se almacena en variables asociadas a cada entorno de despliegue. Las variables de entorno se pueden cambiar fácilmente entre despliegues sin la necesidad de alterar el código fuente.

B.2.4. Servicios de apoyo (*Backing services*)

Los servicios de apoyo son servicios que la aplicación consume a través de la red como parte de su operación normal. Algunos ejemplos incluyen almacenes de datos (como *MySQL* o *MongoDB*), *middleware* de mensajería y colas (como *RabbitMQ* o *Beanstalkd*),

servicios SMTP para envío de correo (como Postfix) y sistemas de almacenamiento en memoria inmediata (como *Memcached*).

El código de una aplicación *twelve-factor app* no hace distinciones entre servicios locales y remotos. Para la aplicación, todos ellos son servicios conectados accesibles mediante una URL u otro mecanismo de localización/credenciales almacenados en la configuración. Durante un despliegue, la aplicación debe poder intercambiar, por ejemplo, una base de datos local *MySQL* por una remota manejada por un tercero como *Amazon RDS* sin necesidad de tener que realizar cambios en el código fuente. De las misma manera, debería poder intercambiar el servicio de correo (SMTP) local por el de un tercero como *Postmark*. En ambos casos, lo único que debe cambiar es la configuración de acceso al recurso.

B.2.5. Construcción, publicación, ejecución (*Build, release, run*)

La base de código de una aplicación se transforma en un despliegue mediante las siguientes tres fases:

- *Build*. La fase de construcción transforma una versión del código fuente (*repository commit*) especificada durante el proceso de despliegue en un paquete ejecutable que contiene ejecutables, dependencias y otros recursos.
- *Release*. En la fase de publicación, el paquete generado en la fase anterior (*build*) es combinado con la configuración del despliegue y el resultado está preparado para pasar a la fase de ejecución.
- *Run* o *runtime*. La fase de ejecución es aquella en la que la aplicación es puesta en marcha mediante el arranque de una serie de procesos asociados con una versión o *release* determinada.

La metodología *twelve-factor app* hace una clara distinción entre las fases: *build*, *release* y *run*. Por ejemplo, es imposible realizar cambios en el código durante la fase de ejecución (*runtime*) ya que estos no se pueden propagar hasta la fase de *build*.

Cada *release* debería tener siempre un identificador único, como la fecha en la que se generó (por ejemplo, `2013-02-29T22:12:54`) o el número de una secuencia creciente (como `v102`). No se debe eliminar o alterar una *release* y cualquier cambio necesario debe ser manejado en una nueva *release*.

La fase de *build* es iniciada por los desarrolladores de la aplicación siempre que exista código nuevo que deba ser desplegado. Sin embargo, la fase de *runtime* puede iniciarse automáticamente por varias causas, como el reinicio de un servidor o porque un proceso ha fallado y ha sido lanzado de nuevo por el gestor de procesos.

B.2.6. Procesos (*Processes*)

La aplicación se ejecuta en el entorno de ejecución como uno o más procesos. En el caso más sencillo, el código es un *script* independiente, el entorno de ejecución es la máquina del programador (con el entorno de programación) y el proceso se lanza mediante el intérprete de comandos (por ejemplo, `ruby my_script.rb`). El caso contrario, un despliegue en producción de una aplicación sofisticada puede utilizar muchos tipos de procesos asociados con cero o más procesos en ejecución.

Los procesos de una aplicación *twelve-factor app* son sin estado (*stateless*) y no comparten información. Cualquier dato que deba persistir debe ser almacenado en un servicio auxiliar con estado, habitualmente una base de datos.

El espacio de memoria y el espacio en disco disponible para el proceso puede ser utilizado como una memoria inmediata para una sola transacción. Por ejemplo, descargar un archivo de gran tamaño, hacer una operación sobre él y almacenar los resultados en una base de datos. *Twelve-factor app* nunca asume que algo que estuvo en memoria o en disco estará disponible para la siguiente transacción.

Algunos sistemas web o *frameworks* dependen de *sticky sessions*, un mecanismo por el cual se almacena información del cliente en la memoria del proceso y se espera que las siguientes peticiones sean encaminadas al mismo proceso. Este sistema es una violación de la metodología y debe ser evitado. Los datos de una sesión son un buen candidato para ser almacenados en una base de datos temporal como *Memcached* o *Redis*.

B.2.7. Asociación de puertos (*Port binding*)

En ocasiones las aplicaciones web son ejecutadas dentro de entornos de ejecución especiales denominados contenedores web. Por ejemplo, una aplicación escrita en PHP podría ejecutarse como un módulo dentro de *Apache HTTP Server* (`mod_php`) y una escrita en *Java* podría hacerlo dentro de una de las múltiples implementaciones de *Servlet* como *Jetty*, *Tomcat*, etc.

Una aplicación *twelve-factor app* es completamente autónoma y no requiere la inyección de ningún elemento en el entorno de ejecución para poder producir un servicio web. La aplicación exporta el protocolo *HTTP* como un servicio estableciendo una asociación con un puerto y recibe las peticiones que llegan a ese puerto. En un entorno de desarrollo local, el programador se conecta a una URL como `http://localhost:5000/` para acceder al servicio exportado por la aplicación. En un entorno de despliegue como test o producción, una capa de encaminamiento (*routing layer*) es la encargada de encaminar las peticiones desde el punto de acceso al público hasta el puerto ligado a los procesos web de la aplicación.

Esto se implementa declarando dentro de la aplicación una dependencia con algún ser-

vidor web, como *Tornado* en *Python*, *Thin* en *Ruby* o *Jetty* en *Java* y otros lenguajes compatibles con *JVM*. Esto tiene lugar en el espacio de usuario, es decir, dentro del código de la aplicación. El contrato con el entorno de ejecución es la conexión con un puerto para atender peticiones.

El protocolo o servicio HTTP no es el único que puede ser exportado mediante asociación de puertos. Casi cualquier tipo de servicio puede ser ejecutado mediante la asociación de un proceso con un puerto dejándolo a la espera de peticiones. Algunos ejemplos incluyen *ejabberd* (protocolo *XMPP*) y *Redis* (protocolo *Redis*).

Es importante recordar que el mecanismo de asociación de puertos permite que una aplicación pueda ser el servicio auxiliar de otra. Para ello, la URL de acceso a la aplicación auxiliar debe estar dada de alta en la configuración de la aplicación consumidora.

B.2.8. Concurrencia (*Concurrency*)

Cualquier programa, una vez ejecutado, es representado por uno o más procesos. Las aplicaciones web han tomado diferentes representaciones de proceso-ejecución. Por ejemplo, los procesos PHP son ejecutados como procesos hijos de Apache, y son arrancados según se necesita por el volumen de peticiones. Los procesos Java utilizan un enfoque opuesto. La máquina virtual Java lanza un proceso masivo que reserva grandes bloques de recursos del sistema (memoria y procesador) durante el arranque, y la concurrencia se maneja internamente mediante *threads*. En ambos casos, los procesos en ejecución son apenas visibles para el desarrollador.

En las aplicaciones *twelve-factor app*, los procesos son objetos (*first-class citizen*). Los procesos en esta metodología están fuertemente inspirados en el modelo UNIX de procesos para ejecutar demonios. Mediante este modelo, el desarrollador puede diseñar la arquitectura de la aplicación para manejar distintos tipos de carga asignando cada tipo de trabajo a un tipo de proceso. Por ejemplo, las peticiones HTTP pueden ser manejadas por procesos web y los trabajos largos de segundo plano (*background tasks*), pueden ser manejadas por procesos worker.

Este modelo de procesos es realmente efectivo cuando se trata de escalar de forma horizontal. En las aplicaciones *twelve-factor app* los procesos no comparten datos y cada uno se especializa en resolver una determinada tarea lo cual hace que añadir más concurrencia sea una tarea sencilla. El conjunto de tipos de procesos y el número de cada uno de ellos en ejecución se denomina *process formation*.

Los procesos de una aplicación *twelve-factor app* nunca deben ser ejecutados como demonios o escribir ficheros de proceso (*PID files*). En su lugar, deben confiar en el gestor de procesos del sistema operativo (como *Upstart*, un gestor de procesos distribuido para

una plataforma en la nube o una herramienta como *Foreman* para entornos locales de desarrollo) que maneje los flujos de datos de salida, responda ante procesos que fallan y atienda las acciones del usuario para realizar arranques y paradas.

B.2.9. Desechabilidad

Los procesos que conforman una aplicación *twelve-factor app* son desechables, lo que significa que pueden ser arrancados y parados sin previo aviso. Esto proporciona las siguientes ventajas:

- Facilita el *escalado* elástico y rápido.
- Despliegues rápidos de código y cambios en la configuración.
- Robustez de los despliegues en producción.

Los procesos que componen una aplicación deben tratar de iniciarse lo más rápidamente posible. Idealmente, a un proceso le lleva unos pocos segundos estar listo para realizar su trabajo desde el momento en el que es invocado. Los tiempos de inicio cortos favorecen un mejor funcionamiento de la aplicación ya que es más fácil afrontar incrementos de carga (*scale up*) y el gestor de procesos puede moverlos más rápidamente entre diferentes ubicaciones físicas distintas para mejorar su rendimiento siempre que sea necesario.

Cuando el gestor procesos envía la señal `SIGTERM` a un proceso, este muere. Para un proceso web, una salida correcta se consigue dejando de escuchar en el puerto de servicio (rechazando nuevas peticiones) y permitiendo que las que están en proceso puedan ser atendidas y finalmente muere. De forma implícita en este modelo es que las peticiones HTTP deben ser cortas (no más de unos pocos segundos). En el caso de peticiones largas periódicas (polling) el cliente debe ser capaz de conectar de nuevo cuando la conexión se interrumpe.

También los procesos deben ser resistentes frente a muertes súbitas, por ejemplo en caso de un fallo en la capa de *hardware*. Una forma de afrontar este problema es utilizar algún mecanismo de colas robusto como *Beanstalkd* que devuelve los trabajos a la cola cuando los clientes se desconectan o no contestan pasado un tiempo (*timeout*).

B.2.10. Paridad Dev/Prod (*Dev/Prod parity*)

Históricamente, siempre han existido diferencias sustanciales entre los entornos de desarrollo (un programador que hace cambios sobre su propia copia local de la aplicación) y producción (una aplicación desplegada y siendo accedida por usuarios). Estas diferencias se manifiestan en tres áreas:

- Tiempo (*time gap*). Un programador trabaja sobre código que puede tardar días, semanas o incluso meses en llegar a producción.

- Puesto (*personnel gap*). Los programadores escriben el código mientras que los ingenieros de operaciones lo despliegan.
- Herramientas (*tools gap*). El *stack* o conjunto de herramientas utilizado por los desarrolladores (*Nginx*, *SQLite*, *OS X* puede ser muy distinto al utilizado en producción (*Apache*, *MySQL* o *PostgreSQL*, *Linux*).

La metodología *twelve-factor app* está diseñada para poder realizar despliegue continuado (*continuous deployment*) reduciendo las diferencias entre desarrollo y producción al mínimo. Así:

- El código escrito por un programador puede estar desplegado en cuestión de horas o incluso unos pocos minutos.
- Los programadores están muy involucrados en el proceso de despliegue y evaluación del comportamiento de la aplicación en producción.
- El entorno de programación y producción es tan parecido como sea posible.

Es importante asegurar también paridad en los servicios auxiliares como la base de datos de la aplicación, su sistema de colas de mensajes o de memoria inmediata (cache). La mayoría de los lenguajes de programación ofrecen bibliotecas y adaptadores que simplifican el acceso a varios de estos servicios. Algunos ejemplos se indican a continuación:

Tipo	Lenguaje	Biblioteca	Adaptador
Base de datos	Ruby	`ActiveRecord`, `DataMapper`	MySQL, PostgreSQL, SQLite
Mensajería	Python	`Celery`	RabbitMQ, Beanstalkd, Redis
Cache	Ruby	`ActiveSupport::Cache`	Memory, filesystem, Memcached

Tabla B.1: Bibliotecas y adaptadores para servicios auxiliares

En ocasiones los desarolladores encuentran muy práctico utilizar servicios auxiliares ligeros para sus entornos de programación locales mientras que para producción se emplean otros más robustos. Por ejemplo, utilizan *SQLite* en desarrollo y *PostgreSQL* en producción.

Sin embargo, la metodología *twelve-factor app* desaconseja el uso de diferentes servicios auxiliares para desarrollo y producción, incluso aunque se usen adaptadores que abstraen las posibles diferencias entre ellos. Cualquier mínima diferencia entre los diferentes entornos pueden provocar pequeñas (o grandes) incompatibilidades y código que funcionaba y pasaba los tests correctamente en los entornos de desarrollo y test falla en producción. Este tipo de errores desincentivan el despliegue continuo. El coste de estas fricciones y la debilitación del despliegue continuo es extremadamente alto cuando se tienen en cuenta a

lo largo de la vida de la aplicación.

Los servicios ligeros ya no son tan atractivos como hace tiempo. Los actuales servicios modernos como *Memcached*, *PostgreSQL* y *RabbitMQ* no son difíciles de instalar ni de ejecutar gracias a los sistemas de distribución de paquetes como *Homebrew*, *Macports* o *APT*. Alternativamente es posible utilizar herramientas de integración como *Chef* o *Puppet* que combinados con entornos virtuales ligeros como los proporcionados por *Vagrant* permiten a los desarrolladores ejecutar en local (desarrollo) con unas características casi idénticas a las de producción. El coste de instalación y uso de estos sistemas es bajo comparado con los beneficios obtenidos cuando existe paridad entre desarrollo y producción y es posible realizar despliegue continuo.

Los adaptadores para los diferentes servicios auxiliares siguen siendo muy útiles pues hacen que el cambio entre unos servicios y otros sean relativamente sencillos. Sin embargo, todos los despliegues de la aplicación (desarrollo, test y producción) debe utilizar el mismo tipo y versión de cada uno de los servicios auxiliares.

B.2.11. Registros (*Logs*)

Los registros o *logs* proporcionan visibilidad sobre el comportamiento de una aplicación en ejecución. Están formados por una recopilación de los flujos de datos de salida (*output streams*) de todos los procesos de que se compone una aplicación y de sus servicios auxiliares. Los *logs* no tienen comienzo ni fin fijos; simplemente existen mientras la aplicación siga en ejecución.

En una aplicación *twelve-factor app* nunca hay que preocuparse por el encaminamiento o almacenamiento de su flujo de datos de salida (*output stream*). No debe intentar escribir o manipular los *logs*. En su lugar, cada proceso en ejecución debe escribir el flujo de eventos en la salida estándar o `STDOUT`. Así, durante las fases de desarrollo, el programador podrá ver el flujo de eventos en la consola que utiliza para arrancar la aplicación y observar su comportamiento.

Cuando la aplicación es desplegada en los entornos de pruebas y producción, los flujos de datos de cada proceso son capturados por el entorno de ejecución, agrupados junto con otros flujos de datos de la aplicación y encaminados hacia uno o varios destinos para su visualización, almacenamiento y archivado. Estos destinos no son directamente visibles o ajustables por la aplicación si no que están completamente manejados por el entorno de ejecución. Para realizar esta tarea se utilizan herramientas *open source* como *Logplex* y *Fluent*.

El flujo de datos de eventos de una aplicación puede ser encaminado hacia un fichero o visualizado en tiempo real mediante el comando `tail` en una consola. Y más importante, puede ser enviado a un sistema de indexado y análisis como *Splunk*, o a un almacén de datos de propósito general como *Hadoop/Hive*. Estos sistemas ofrecen una gran potencia

y flexibilidad para examinar el comportamiento de una aplicación a lo largo del tiempo como:

- Encontrar eventos específicos ocurridos en el pasado.
- Obtención de gráficos de tendencias (como el número de peticiones por minuto).
- Activación de alertas de acuerdo a heurísticas definidas por el usuario.

B.2.12. Procesos de administración (*Admin processes*)

La alineación de procesos (*process formation*) es el conjunto de procesos utilizados para implementar la lógica de negocio de la aplicación. De forma separada, es habitual que los desarrolladores necesiten ejecutar puntualmente tareas administrativas y de mantenimiento (*one-off processes*) como:

- Ejecutar una migración en la base de datos. Por ejemplo: `rake db:migrate` en *Rails*.
- Lanzar una consola (*REPL shell*) para ejecutar comandos que permitan inspeccionar los modelos de una aplicación en ejecución. La mayoría de los lenguajes proporcionan una consola cuando se invoca el intérprete sin argumentos.
- Ejecutar un *script* puntual existente en el repositorio de la aplicación. Por ejemplo: `ruby scripts/fix_bad_records.rb`.

Las tareas de administración (*one-off processes*) deben ejecutarse en condiciones idénticas (misma base de código y configuración) que las de los procesos que conforman la aplicación. Por otro lado, se deben aplicar las mismas técnicas para salvaguardar el aislamiento de dependencias que se utilizan para ejecutar la aplicación. Por ejemplo, si un proceso web *Ruby* se inicia utilizando el comando `bundle exec thin start`, una migración de base de datos debería ejecutarse mediante el comando `bundle exec rake db:migrate`.

Bibliografía

[1] Google Inc., "Zinkia Entertainment mejora en seguridad y productividad gracias a Google Apps." http://goo.gl/XVyeh, 2011.

[2] José Olalla, "BBVA apuesta por Google Apps." http://goo.gl/Ru38w, 2012.

[3] Google Inc., "El caso de éxito de Fon." http://goo.gl/yu5uv, 2011.

[4] P. Mell and T. Grance, "The NIST definition of cloud computing," *National Institute of Standards and Technology*, 2011.

[5] "*Secure Cloud and SaaS Based Treasury Management Services*." http://www.kyriba.com/solutions/cloud-saas-based-treasury-management-services.

[6] "*Sales Cloud: Sales Force Automation Tools*." http://www.salesforce.com/sales-cloud/overview/.

[7] "*Customer Service Software and Support Software Service Cloud*." http://www.salesforce.com/service-cloud/overview/.

[8] "*Enterprise Social Network Software*." https://www.yammer.com/product/.

[9] "*Online Invoicing, Accounting and Billing Software*." http://www.freshbooks.com.

[10] "*Ruby On Rails and PHP Cloud Hosting PaaS*." https://www.engineyard.com.

[11] "*Ruby Programming Language*." http://www.ruby-lang.org/en/.

[12] "*PHP: Hypertext Preprocessor*." http://php.net.

[13] "*Node.js*." http://nodejs.org.

[14] "*Engine Yard Announces General Availability of Node.js*." https://www.engineyard.com/company/press/2012-08-21-engine-yard-announces-general-availability-of-nodejs-on-engine-yard-cloud-the-industry-leading-platform-as-a-service.

[15] "*Heroku Cloud Application Platform*." http://www.heroku.com.

[16] "*Heroku Buildpacks*." https://devcenter.heroku.com/articles/buildpacks.

[17] "*Ruby Buildpack*." https://github.com/heroku/heroku-buildpack-ruby.

[18] "*Node.js Buildpack*." https://github.com/heroku/heroku-buildpack-nodejs.

[19] "*Clojure Buildpack*." https://github.com/heroku/heroku-buildpack-clojure.

[20] "*Python Buildpack*." https://github.com/heroku/heroku-buildpack-python.

[21] "*Java Buildpack*." https://github.com/heroku/heroku-buildpack-java.

[22] "*Gradle Buildpack*." https://github.com/heroku/heroku-buildpack-gradle.

[23] "*Grails Buildpack*." https://github.com/heroku/heroku-buildpack-grails.

[24] "*Scala Buildpack*." https://github.com/heroku/heroku-buildpack-scala.

[25] "*Play Buildpack*." https://github.com/heroku/heroku-buildpack-play.

[26] "*Common Lisp Buildpack*." https://github.com/mtravers/heroku-buildpack-cl.

[27] "*Elixir Buildpack*." https://github.com/goshakkk/heroku-buildpack-elixir.

[28] "*Erlang Buildpack*." https://github.com/archaelus/heroku-buildpack-erlang.

[29] "*Go Buildpack*." https://github.com/kr/heroku-buildpack-go.

[30] "*JRuby Buildpack*." https://github.com/jruby/heroku-buildpack-jruby.

[31] "*Lua Buildpack*." https://github.com/leafo/heroku-buildpack-lua.

[32] "*Perl/PSGI Buildpack*." https://github.com/miyagawa/heroku-buildpack-perl.

[33] "*Apache2/mod_perl Buildpack*." https://github.com/lstoll/heroku-buildpack-perl.

[34] "*R Buildpack*." https://github.com/virtualstaticvoid/heroku-buildpack-r.

[35] "*Silex Buildpack*." https://github.com/klaussilveira/heroku-buildpack-silex.

[36] "*Auto-scaling Platform as a Service for applications*." https://openshift.redhat.com.

[37] "*Heroku Postgres (Heroku Add-on)*." https://addons.heroku.com/heroku-postgresql.

[38] "*JustOneDB (Heroku Add-on)*." https://addons.heroku.com/justonedb.

[39] "*ClearDB MySQL Database (Heroku Add-on)*." https://addons.heroku.com/cleardb.

[40] "*Amazon RDS (Heroku Add-on)*." https://addons.heroku.com/amazon_rds.

[41] "*Xeround Cloud Database (Heroku Add-on)*." https://addons.heroku.com/xeround.

[42] "*Rack - Ruby Webserver Interface*." http://rack.github.com.

[43] "*AWS Elastic Beanstalk*." http://aws.amazon.com/elasticbeanstalk.

[44] "*Google App Engine*." https://developers.google.com/appengine/.

[45] "*Windows Azure: Microsoft's Cloud Platform*." http://www.windowsazure.com.

[46] "*Declaring and Scaling Process Types with Procfile*." https://devcenter.heroku.com/articles/procfile.

[47] "*Foreman*." https://github.com/ddollar/foreman.

[48] "Artículo sobre *Logplex*." https://devcenter.heroku.com/articles/logplex.

[49] "*Oink (Memory profiling tool for Ruby)*." https://github.com/noahd1/oink.

[50] "*Heapy (Memory profiling tool for Python)*." http://guppy-pe.sourceforge.net/#Heapy.

[51] "*Linux Containers (LXC)*." http://lxc.sourceforge.net.

[52] "*Heroku Postgres (Database as a Service)*." https://postgres.heroku.com.

[53] "*MongoDB*." http://www.mongodb.org.

[54] "Información sobre *10gen*." http://www.10gen.com.

[55] "*GridFS - MongoDB*." http://docs.mongodb.org/manual/applications/gridfs/.

[56] "*MapReduce - MongoDB*." http://docs.mongodb.org/manual/applications/map-reduce/.

[57] "Planes y precios del servicio *MongoHQ*." https://www.mongohq.com/pricing.

[58] "*Amazon Simple Storage Service (Amazon S3)*." http://aws.amazon.com/s3/.

[59] "*Amazon S3 Regions*." http://aws.amazon.com/s3/faqs/#Where_is_my_data_stored.

[60] "*BitTorrent* y *Amazon S3*." http://aws.amazon.com/s3/faqs/#What_is_the_BitTorrent_TM_protocol_and_how_do_I_use_it_with_Amazon_S3.

[61] "*Amazon S3 Service Level Agreement*." http://aws.amazon.com/s3-sla/.

[62] "*Amazon AWS Simple Monthly Calculator*." http://aws.amazon.com/calculator/.

[63] "*Amazon S3 SDK for Android*." http://aws.amazon.com/sdkforandroid/.

[64] "*Amazon S3 SDK for iOS*." http://aws.amazon.com/sdkforios/.

[65] "*Amazon S3 SDK for Java*." http://aws.amazon.com/sdkforjava/.

[66] "*Amazon S3 SDK for .NET*." http://aws.amazon.com/sdkfornet/.

[67] "*Amazon S3 SDK for PHP*." http://aws.amazon.com/sdkforphp/.

[68] "*Amazon S3 SDK for Ruby*." http://aws.amazon.com/sdkforruby/.

[69] "*Google Chart Tools*." https://developers.google.com/chart/terms.

[70] "*Google Chart Tools - Terms*." https://developers.google.com/chart/terms.

[71] "*Flickr*." http://www.flickr.com.

[72] "*Ruby Language*." http://www.ruby-lang.org.

[73] "*Sinatra*." http://www.sinatrarb.com.

[74] "*Rails Web Framework*." http://rubyonrails.org.

[75] "*Ramaze Web Framework*." http://ramaze.net.

[76] "*Merb Web Framework*." http://www.merbivore.com.

[77] "*GitHub*." https://github.com.

[78] "*BBC - Homepage*." http://www.bbc.co.uk.

[79] "*Apple*." http://www.apple.com.

[80] "*Express - node.js web application framework*." http://expressjs.com.

[81] "*Nancy - Lightweight Web Framework for .NET*." http://nancyfx.org.

[82] "*Mercury*." https://github.com/nrk/mercury.

[83] S. Allamaraju, *RESTful Web Services Cookbook*. O'Reilly Media, 2010.

[84] "RFC 2617 *Basic and Digest Access Authentication*." http://www.ietf.org/rfc/rfc2617.txt.

[85] "*MongoMapper*." http://mongomapper.com.

[86] "*Mongoid*." http://mongoid.org/en/mongoid/index.html.

[87] "*Mongo ODM*." https://github.com/carlosparamio/mongo_odm.

[88] "*MongoModel*." http://www.mongomodel.org.

[89] "*Heroku Toolbelt*." https://toolbelt.heroku.com.

[90] "*Heroku Sign Up*." https://api.heroku.com/signup.

[91] "*Httperf*." http://www.hpl.hp.com/research/linux/httperf/.

[92] "*Apache HTTP server benchmarking tool*." http://httpd.apache.org/docs/2.2/programs/ab.html.

[93] "*curl-loader*." http://curl-loader.sourceforge.net.

[94] "*Siege*." http://www.joedog.org/siege-home/.

[95] "*New Relic - Web Application Performance Management*." http://newrelic.com.

[96] A. Harris and K. Haase, *Sinatra: Up and Running.* O'Reilly Media, 2011.

[97] M. P. McGrath, *Understanding PaaS.* O'Reilly Media, 2012.

[98] "*MongoMapper Documentation*." http://mongomapper.com/documentation/.

Glosario

Benchmarking Técnica para evaluar el rendimiento de un sistema mediante la comparación de varias pruebas realizadas bajo distintas condiciones.

Hardware Partes tangibles de un sistema informático como la placa base, memoria, fuente de alimentación, unidad de estado sólido y disco duro entre otros.

Middleware Elemento *software* que sirve de adaptador o mecanismo de unión para otras aplicaciones, redes, *hardware* o sistemas operativos.

Multi-tenant Arquitectura de *software* donde una sola instancia de una aplicación proporciona servicio a varios clientes pertenecientes a organizaciones distintas.

Software Componentes lógicos (sistemas operativos y aplicaciones) de un sistema informático que permiten la realización de tareas específicas.

Swap Área de memoria secundaria de un sistema (normalmente disco duro o de estado sólido) dedicada a almacenar páginas de memoria que han sido desalojadas de la memoria principal (RAM).

APM *(Application Performance Management)* Disciplina dentro de la administración de sistemas que se especializa en el control del rendimiento y la disponibilidad de aplicaciones *software*.

Autenticar Mecanismo mediante el cual un sistema es capaz de determinar la identidad de sus usuarios.

Autorizar Mecanismo mediante el cual un sistema es capaz de determinar qué nivel de acceso debe tener a recursos protegidos un usuario previamente autenticado.

Código Abierto *(Open Source)* Término con el que se conoce al *software* desarrollado y distribuido libremente.

DNS *(Domain Name System)* Sistema que permite el nombrado jerárquico de máquinas, servicios o cualquier otro recurso conectado a Internet o a un red privada. Entre sus funciones destaca la responsabilidad de determinar, mediante registros `MX`, las máquinas responsables de procesar el correo electrónico para un determinado dominio así como de enumerar máquinas de *backup* y su prioridad.

DSL ***(Domain Specific Language)*** Se denominan así aquellos lenguajes concebidos para resolver un determinado tipo de problemas. Por ejemplo, el lenguaje SQL está diseñado específicamente para realizar consultas sobre bases de datos relacionales.

Gema ***(Gem)*** Formato estándar producido por la herramienta *RubyGems* que permite la distribución de programas y bibliotecas escritas en *Ruby*.

HTTP ***(Hypertext Transfer Protocol)*** Protocolo de aplicación que permite el intercambio de documentos hipermedia en Internet.

IMAP ***(Internet Message Access Protocol)*** Protocolo moderno que permite el acceso a correo electrónico almacenado en un servidor remoto.

JSON ***(JavaScript Object Notation)*** Es un formato de texto abierto estándar inteligible por humanos diseñado para el intercambio de datos. Esta derivado de la representación de objetos del lenguaje *JavaScript*.

Latencia Medida del tiempo de retardo experimentado en un sistema.

Migración (Base de datos) Mecanismo mediante el cual es posible alterar la estructura de una base de datos de una manera controlada y organizada.

ORM ***(Object-Relational Mapping)*** Técnica de programación que trata de eliminar la incompatibilidad de tipos de datos entre lenguajes de programación y bases de datos relacionales.

POP ***(Post Office Protocol)*** Protocolo que permite la obtención de correo electrónico almacenado en un servidor remoto.

RFC ***(Request For Comments)*** Documentos especiales escritos y publicados por individuos comprometidos con el desarrollo y mantenimiento de Internet. Tienen el importante propósito de servir de documentación para nuevos desarrollos tecnológicos y estándares en la Red.

RIA ***(Rich Internet Application)*** Aplicaciones que combinan las características más ventajosas de las aplicaciones web (como su fácil distribución) y las aplicaciones de escritorio (como un mejor rendimiento).

SMTP ***(Simple Mail Transfer Protocol)*** Sistema mediante el cual se transmite correo electrónico en la red Internet.

SSL ***(Secure Sockets Layer)*** Estándar abierto propuesto para el establecimiento de un canal seguro de comunicaciones que impida la interceptación de información crítica como pueden ser contraseñas, datos de tarjetas de crédito, médicos o personales entre otros.

TLS ***(Transport Layer Security)*** Protocolo de seguridad para comunicaciones creado por la IETF (*Internet Engineering Task Force*) que toma como base SSL (*Secure Sockets Layer)* y otros protocolos.

UNIX Sistema operativo portable, multitarea y multiusuario desarrollado en los años setenta por un grupo de empleados de los laboratorios *Bell*. Algunos sistemas operativos actuales como *Linux* u *OS X* comparten varios aspectos de diseño con él.

URL ***(Uniform Resource Locator)*** Sistema unificado de identificación de recursos en la red. Es el modo estándar de proporcionar la dirección de cualquier recurso en Internet.

XML ***(Extensible Markup Language)*** Lenguaje de marcado derivado de SGML utilizado para definir documentos estructurados.